AVR-LibC

2.2.0

Generated by Doxygen 1.9.6

1 AVR-LibC

1.1 Introduction

The latest version of this document is always available from `https://avrdudes.github.io/avr-libc/`

This documentation is distributed under the same licensing conditions as the entire library itself, see License below.

The AVR-LibC package provides a subset of the standard C library for `Microchip (formerly Atmel) AVR 8-bit RISC microcontrollers`. In addition, the library provides the basic startup code needed by most applications.

There is a wealth of information in this document which goes beyond simply describing the interfaces and routines provided by the library. We hope that this document provides enough information to get a new AVR developer up to speed quickly using the freely available development tools: binutils, gcc, AVR-LibC and many others.

If you find yourself stuck on a problem which this document doesn't quite address, you may wish to post a message to the avr-gcc mailing list. Most of the developers of the AVR Binutils and GCC ports in addition to the devleopers of AVR-LibC subscribe to the list, so you will usually be able to get your problem resolved. You can subscribe to the list at `http://lists.nongnu.org/mailman/listinfo/avr-gcc-list` . Before posting to the list, you might want to try reading the Frequently Asked Questions chapter of this document.

Note

If you think you've found a bug, or have a suggestion for an improvement, either in this documentation or in the library itself, please use the `bug tracker` to ensure the issue won't be forgotten.

1.2 General Information about this Library

In general, it has been the goal to stick as best as possible to established standards while implementing this library. Commonly, this refers to the C library as described by the ANSI X3.159-1989 and ISO/IEC 9899:1990 ("ANSI-C") standard, as well as parts of their successor ISO/IEC 9899:1999 ("C99"). Some additions have been inspired by other standards like IEEE Std 1003.1-1988 ("POSIX.1"), while other extensions are purely AVR-specific (like the entire program-space string interface).

Unless otherwise noted, functions of this library are *not* guaranteed to be reentrant. In particular, any functions that store local state are known to be non-reentrant, as well as functions that manipulate I/O registers like the EEPROM access routines. If these functions are used within both standard and interrupt contexts undefined behaviour will result. See the FAQ for a more detailed discussion.

1.3 Supported Devices

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

megaAVR Devices: ATmega103, ATmega128, ATmega128A, ATmega1280, ATmega1281, ATmega1284, ATmega1284P, ATmega16, ATmega161, ATmega162, ATmega163, ATmega164A, ATmega164P, ATmega164PA, ATmega165, ATmega165A, ATmega165P, ATmega165PA, ATmega168, ATmega168A, ATmega168P, ATmega168PA, ATmega168PB, ATmega16A, ATmega2560, ATmega2561, ATmega32, ATmega32A, ATmega323, ATmega324A, ATmega324P, ATmega324PA, ATmega324PB, ATmega325, ATmega325A, ATmega325P, ATmega325PA, ATmega3250, ATmega3250A, ATmega3250P, ATmega3250PA, ATmega328, ATmega328P, ATmega328PB, ATmega48, ATmega48A, ATmega48PA, ATmega48PB, ATmega48P, ATmega64, ATmega64A, ATmega640, ATmega644, ATmega644A, ATmega644P, ATmega644PA, ATmega645, ATmega645A, ATmega645P, ATmega6450, ATmega6450A, ATmega6450P, ATmega8, ATmega8A, ATmega88, ATmega88A, ATmega88P, ATmega88PA, ATmega88PB, ATmega8515, ATmega8535

megaAVR 0-Series Devices: ATmega808, ATmega809, ATmega1608, ATmega1609, ATmega3208, ATmega3209, ATmega4808, ATmega4809

tinyAVR Devices: ATtiny11 [1], ATtiny12 [1], ATtiny13, ATtiny13A, ATtiny15 [1], ATtiny22, ATtiny24, ATtiny24A, ATtiny25, ATtiny26, ATtiny261, ATtiny261A, ATtiny28 [1], ATtiny2313, ATtiny2313A, ATtiny4313, ATtiny43U, ATtiny44, ATtiny44A, ATtiny441, ATtiny45, ATtiny461, ATtiny461A, ATtiny48, ATtiny828, ATtiny84, ATtiny84A, ATtiny841, ATtiny85, ATtiny861, ATtiny861A, ATtiny88, ATtiny1634

tinyAVR 0-Series Devices: ATtiny202, ATtiny204, ATtiny402, ATtiny404, ATtiny406, ATtiny804, ATtiny806, ATtiny807, ATtiny1604, ATtiny1606, ATtiny1607

tinyAVR 1-Series Devices: ATtiny212, ATtiny214, ATtiny412, ATtiny414, ATtiny416, ATtiny417, ATtiny814, ATtiny816, ATtiny817, ATtiny1614, ATtiny1616, ATtiny1617, ATtiny3214, ATtiny3216, ATtiny3217

tinyAVR 2-Series Devices: ATtiny424, ATtiny426, ATtiny427, ATtiny824, ATtiny826, ATtiny827, ATtiny1624, ATtiny1626, ATtiny1627, ATtiny3224, ATtiny3226, ATtiny3227

Reduced tinyAVR Devices with only 16 GPRs: ATtiny4, ATtiny5, ATtiny9, ATtiny10, ATtiny102, ATtiny104, ATtiny20, ATtiny40

Automotive AVR Devices: ATtiny87, ATtiny167, ATA5505, ATA5272, ATA5702M322, ATA5782, ATA5790, ATA5790N, ATA5831, ATA5795, ATA6285, ATA6286, ATA6289, ATA6612C, ATA6613C, ATA6614Q, ATA6616C, ATA6617C, ATA664251

Automotive CAN AVR Devices: ATmega16M1, ATmega32C1, ATmega32M1, ATmega64C1, ATmega64M1

CAN AVR Devices: AT90CAN32, AT90CAN64, AT90CAN128

LCD AVR Devices: ATmega169, ATmega169A, ATmega169P, ATmega169PA, ATmega329, ATmega329A, ATmega329P, ATmega329PA, ATmega3290, ATmega3290A, ATmega3290P, ATmega3290PA, ATmega649, ATmega649A, ATmega6490, ATmega6490A, ATmega6490P, ATmega649P

Lighting AVR Devices: AT90PWM1, AT90PWM2, AT90PWM2B, AT90PWM216, AT90PWM3, AT90PWM3B, AT90PWM316, AT90PWM161, AT90PWM81

Smart Battery AVR Devices: ATmega8HVA, ATmega16HVA, ATmega16HVA2, ATmega16HVB, ATmega16HVBREVB, ATmega32HVB, ATmega32HVBREVB, ATmega64HVE, ATmega64HVE2, ATmega406

USB AVR Devices: AT76C711 [3], AT90USB82, AT90USB162, AT90USB646, AT90USB647, AT90USB1286, AT90USB1287, ATmega8U2, ATmega16U2, ATmega16U4, ATmega32U2, ATmega32U4, ATmega32U6, AT43USB320, AT43USB355

XMEGA Devices: ATxmega8E5, ATxmega16A4, ATxmega16D4, ATxmega32A4, ATxmega32D3, ATxmega32D4, ATxmega32E5, ATxmega64A1, ATxmega64A3, ATxmega64D3, ATxmega64D4, ATxmega128A1, ATxmega128A3, ATxmega128D3, ATxmega128D4, ATxmega192A3, ATxmega192D3, ATxmega256A3, ATxmega256A3B, ATxmega256D3

USB XMEGA Devices: ATxmega16A4U, ATxmega16C4, ATxmega32A4U, ATxmega32C3, ATxmega32C4, ATxmega64A1U, ATxmega64A3U, ATxmega64A4U, ATxmega64B1, ATxmega64B3, ATxmega64C3, ATxmega128A1U, ATxmega128A3U, ATxmega128A4U, ATxmega128B1, ATxmega128B3, ATxmega128C3, ATxmega192A3U, ATxmega192C3, ATxmega256A3U, ATxmega256A3BU, ATxmega256C3, ATxmega384C3, ATxmega384D3

AVR-Dx Devices: AVR16DD14, AVR16DD20, AVR16DD28, AVR16DD32, AVR32DA28, AVR32DA32, AVR32DA48, AVR32DB28, AVR32DB32, AVR32DB48, AVR32DD14, AVR32DD20, AVR32DD28, AVR32DD32, AVR64DA28, AVR64DA32, AVR64DA48, AVR64DA64, AVR64DB28, AVR64DB32, AVR64DB48, AVR64DB64, AVR64DD14, AVR64DD20, AVR64DD28, AVR64DD32, AVR128DA28, AVR128DA32, AVR128DA48, AVR128DA64, AVR128DB28, AVR128DB32, AVR128DB48, AVR128DB64

USB AVR-Dx Devices: AVR64DU28, AVR64DU32

AVR-Ex Devices: AVR16EA28, AVR16EA32, AVR16EA48, AVR16EB14, AVR16EB20, AVR16EB28, AVR16EB32, AVR32EA28, AVR32EA32, AVR32EA48, AVR64EA28, AVR64EA32, AVR64EA48

Wireless AVR Devices: ATmega644RFR2, ATmega64RFR2, ATmega128RFA1, ATmega1284RFR2, ATmega128RFR2, ATmega2564RFR2, ATmega256RFR2

Miscellaneous Devices: AT94K [2], AT86RF401, AT90SCR100, M3000 [4]

Classic AVR Devices: AT90S1200 [1], AT90S2313, AT90S2323, AT90S2333, AT90S2343, AT90S4414, AT90S4433, AT90S4434, AT90S8515, AT90C8534, AT90S8535

Note [1] Assembly only. There is no direct support for these devices to be programmed in C since they do not have a RAM based stack. Still, it could be possible to program them in C, see the FAQ for an option.

Note [2] The AT94K devices are a combination of FPGA and AVR microcontroller. [TRoth-2002/11/12: Not sure of the level of support for these. More information would be welcomed.]

Note [3] The AT76C711 is a USB to fast serial interface bridge chip using an AVR core.

Note [4] The M3000 is a motor controller AVR ASIC from Intelligent Motion Systems (IMS) / Schneider Electric.

1.4 AVR-LibC License

AVR-LibC can be freely used and redistributed, provided the following license conditions are met.

```
The contents of AVR-LibC are licensed with a Modified BSD License.

All of this is supposed to be Free Software, Open Source, DFSG-free,
GPL-compatible, and OK to use in both free and proprietary applications.

See the license information in the individual source files for details.

Additions and corrections to this file are welcome.

********************************************************************************
Portions of avr-libc are Copyright (c) 1999-2024
Werner Boellmann,
Dean Camera,
Pieter Conradie,
Brian Dean,
Keith Gudger,
Wouter van Gulik,
Bjoern Haase,
Steinar Haugen,
Peter Jansen,
Reinhard Jessich,
Magnus Johansson,
Georg Johann Lay,
Harald Kipp,
Carlos Lamas,
Cliff Lawson,
Artur Lipowski,
Marek Michalkiewicz,
Todd C. Miller,
Rich Neswold,
Colin O'Flynn,
Bob Paddock,
Andrey Pashchenko,
Reiner Patommel,
Florin-Viorel Petrov,
Alexander Popov,
Michael Rickman,
Theodore A. Roth,
Juergen Schilling,
Philip Soeberg,
Anatoly Sokolov,
Nils Kristian Strom,
Michael Stumpf,
Stefan Swanepoel,
Helmut Wallner,
Eric B. Weddington,
Joerg Wunsch,
Dmitry Xmelkov,
egnite Software GmbH,
The Regents of the University of California.
All rights reserved.

   Redistribution and use in source and binary forms, with or without
   modification, are permitted provided that the following conditions are met:

   * Redistributions of source code must retain the above copyright
     notice, this list of conditions and the following disclaimer.

   * Redistributions in binary form must reproduce the above copyright
     notice, this list of conditions and the following disclaimer in
     the documentation and/or other materials provided with the
     distribution.

   * Neither the name of the copyright holders nor the names of
     contributors may be used to endorse or promote products derived
     from this software without specific prior written permission.

   THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
   AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
   IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
   ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
   LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
   CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
```

2 Toolchain Overview

2.1 Introduction

Welcome to the open source software development toolset for the Microchip (formerly Atmel) AVR!

There is not a single tool that provides everything needed to develop software for the AVR. It takes many tools working together. Collectively, the group of tools are called a toolset, or commonly a toolchain, as the tools are chained together to produce the final executable application for the AVR microcontroller.

The following sections provide an overview of all of these tools. You may be used to cross-compilers that provide everything with a GUI front-end, and not know what goes on "underneath the hood". You may be coming from a desktop or server computer background and not used to embedded systems. Or you may be just learning about the most common software development toolchain available on Unix and Linux systems. Hopefully the following overview will be helpful in putting everything in perspective.

2.2 FSF and GNU

According to its website, "the Free Software Foundation (FSF), established in 1985, is dedicated to promoting computer users' rights to use, study, copy, modify, and redistribute computer programs. The FSF promotes the development and use of free software, particularly the GNU operating system, used widely in its GNU/Linux variant." The FSF remains the primary sponsor of the GNU project.

The GNU Project was launched in 1984 to develop a complete Unix-like operating system which is free software: the GNU system. GNU is a recursive acronym for "GNU's Not Unix"; it is pronounced guh-noo, approximately like canoe.

One of the main projects of the GNU system is the GNU Compiler Collection, or GCC, and its sister project, GNU Binutils. These two open source projects provide a foundation for a software development toolchain. Note that these projects were designed to originally run on Unix-like systems.

2.3 GCC

GCC stands for GNU Compiler Collection. GCC is highly flexible compiler system. It has different compiler front-ends for different languages. It has many back-ends that generate assembly code for many different processors and host operating systems. All share a common "middle-end", containing the generic parts of the compiler, including a lot of optimizations.

In GCC, a *host* system is the system (processor/OS) that the compiler runs on. A *target* system is the system that the compiler compiles code for. And, a *build* system is the system that the compiler is built (from source code) on. If a compiler has the same system for *host* and for *target*, it is known as a *native* compiler. If a compiler has different systems for *host* and *target*, it is known as a cross-compiler. (And if all three, *build*, *host*, and *target* systems are different, it is known as a Canadian cross compiler, but we won't discuss that here.) When GCC is built to execute on a *host* system such as FreeBSD, Linux, or Windows, and it is built to generate code for the AVR microcontroller *target*, then it is a cross compiler, and this version of GCC is commonly known as "AVR GCC". In documentation, or discussion, AVR GCC is used when referring to GCC targeting specifically the AVR, or something that is AVR specific about GCC. The term "GCC" is usually used to refer to something generic about GCC, or about GCC as a whole.

GCC is different from most other compilers. GCC focuses on translating a high-level language to the target assembly only. AVR GCC has three available compilers for the AVR: C language, C++, and Ada. The compiler itself does not assemble or link the final code.

GCC is also known as a "driver" program, in that it knows about, and drives other programs seamlessly to create the final output. The assembler, and the linker are part of another open source project called GNU Binutils. GCC knows how to drive the GNU assembler (gas) to assemble the output of the compiler. GCC knows how to drive the GNU linker (ld) to link all of the object modules into a final executable.

The two projects, GCC and Binutils, are very much interrelated and many of the same volunteers work on both open source projects.

When GCC is built for the AVR target, the actual program names are prefixed with "avr-". So the actual executable name for AVR GCC is: avr-gcc. The name "avr-gcc" is used in documentation and discussion when referring to the program itself and not just the whole AVR GCC system.

See the GCC Web Site and GCC User Manual for more information about GCC.

2.4 GNU Binutils

The name GNU Binutils stands for "Binary Utilities". It contains the GNU assembler (gas), and the GNU linker (ld), but also contains many other utilities that work with binary files that are created as part of the software development toolchain.

Again, when these tools are built for the AVR target, the actual program names are prefixed with "avr-". For example, the assembler program name, for a native assembler is "as" (even though in documentation the GNU assembler is commonly referred to as "gas"). But when built for an AVR target, it becomes "avr-as". Below is a list of the programs that are included in Binutils:

avr-as

The Assembler.

avr-ld

The Linker.

avr-ar

Create, modify, and extract from libraries (archives).

avr-ranlib

Generate index to library (archive) contents.

avr-objcopy

Copy and translate object files to different formats.

avr-objdump

Display information from object files including disassembly.

avr-size

List section sizes and total size.

avr-nm

List symbols from object files.

avr-strings

List printable strings from files.

avr-strip

Discard symbols from files.

avr-readelf

Display the contents of ELF format files.

avr-addr2line

Convert addresses to file and line.

avr-c++filt

Filter to demangle encoded C++ symbols.

2.5 AVR-LibC

GCC and Binutils provides a lot of the tools to develop software, but there is one critical component that they do not provide: a Standard C Library.

There are different open source projects that provide a Standard C Library depending upon your system time, whether for a native compiler (GNU Libc), for some other embedded system (Newlib), or for some versions of Linux (uCLibc). The open source AVR toolchain has its own Standard C Library project: AVR-LibC.

AVR-LibC provides many of the same functions found in a regular Standard C Library and many additional library functions that is specific to an AVR. Some of the Standard C Library functions that are commonly used on a PC environment have limitations or additional issues that a user needs to be aware of when used on an embedded system.

AVR-LibC also contains the most documentation about the whole AVR toolchain.

2.6 Building Software

Even though GCC, Binutils, and AVR-LibC are the core projects that are used to build software for the AVR, there is another piece of software that ties it all together: Make. GNU Make is a program that makes things, and mainly software. Make interprets and executes a Makefile that is written for a project. A Makefile contains dependency rules, showing which output files are dependent upon which input files, and instructions on how to build output files from input files.

Some distributions of the toolchains, and other AVR tools such as MFile, contain a Makefile template written for the AVR toolchain and AVR applications that you can copy and modify for your application.

See the GNU Make User Manual for more information.

2.7 AVRDUDE

After creating your software, you'll want to program your device. You can do this by using the program AVRDUDE which can interface with various hardware devices to program your processor.

AVRDUDE is a very flexible package. All the information about AVR processors and various hardware programmers is stored in a text database. This database can be modified by any user to add new hardware or to add an AVR processor if it is not already listed.

2.8 GDB / Insight / DDD

The GNU Debugger (GDB) is a command-line debugger that can be used with the rest of the AVR toolchain. Insight is GDB plus a GUI written in Tcl/Tk. Both GDB and Insight are configured for the AVR and the main executables are prefixed with the target name: avr-gdb, and avr-insight. There is also a "text mode" GUI for GDB: avr-gdbtui. DDD (Data Display Debugger) is another popular GUI front end to GDB, available on Unix and Linux systems.

2.9 AVaRICE

AVaRICE is a back-end program to AVR GDB and interfaces to the AVR JTAG In-Circuit Emulator (ICE), to provide emulation capabilities.

2.10 SimulAVR

SimulAVR is an AVR simulator used as a back-end with AVR GDB.

2.11 Utilities

There are also other optional utilities available that may be useful to add to your toolset.

`SRecord` is a collection of powerful tools for manipulating EPROM load files. It reads and writes numerous EPROM file formats, and can perform many different manipulations.

`MFile` is a simple Makefile generator is meant as an aid to quickly customize a Makefile to use for your AVR application.

2.12 Toolchain Distributions (Distros)

All of the various open source projects that comprise the entire toolchain are normally distributed as source code. It is left up to the user to build the tool application from its source code. This can be a very daunting task to any potential user of these tools.

Luckily there are people who help out in this area. Volunteers take the time to build the application from source code on particular host platforms and sometimes packaging the tools for convenient installation by the end user. These packages contain the binary executables of the tools, pre-made and ready to use. These packages are known as "distributions" of the AVR toolchain, or by a more shortened name, "distros".

AVR toolchain distros are available on FreeBSD, Windows, Mac OS X, and certain flavors of Linux.

2.13 Open Source

All of these tools, from the original source code in the multitude of projects, to the various distros, are put together by many, many volunteers. All of these projects could always use more help from other people who are willing to volunteer some of their time. There are many different ways to help, for people with varying skill levels, abilities, and available time.

You can help to answer questions in mailing lists such as the avr-gcc-list, or on forums at the AVR Freaks website. This helps many people new to the open source AVR tools.

If you think you found a bug in any of the tools, it is always a big help to submit a good bug report to the proper project. A good bug report always helps other volunteers to analyze the problem and to get it fixed for future versions of the software.

You can also help to fix bugs in various software projects, or to add desirable new features.

Volunteers are always welcome! :-)

3 Memory Areas and Using malloc()

3.1 Introduction

Many of the devices that are possible targets of AVR-LibC have a minimal amount of RAM. The smallest parts supported by the C environment come with 128 bytes of RAM. This needs to be shared between initialized and uninitialized variables (sections .data and .bss), the dynamic memory allocator, and the stack that is used for calling subroutines and storing local (automatic) variables.

Also, unlike larger architectures, there is no hardware-supported memory management which could help in separating the mentioned RAM regions from being overwritten by each other.

The standard RAM layout is to place .data variables first, from the beginning of the internal RAM, followed by .bss. The stack is started from the top of internal RAM, growing downwards. The so-called "heap" available for the dynamic memory allocator will be placed beyond the end of .bss. Thus, there's no risk that dynamic memory will ever collide with the RAM variables (unless there were bugs in the implementation of the allocator). There is still a risk that the heap and stack could collide if there are large requirements for either dynamic memory or stack space. The former can even happen if the allocations aren't all that large but dynamic memory allocations get fragmented over time such that new requests don't quite fit into the "holes" of previously freed regions. Large stack space requirements can arise in a C function containing large and/or numerous local variables or when recursively calling function.

Note

The pictures shown in this document represent typical situations where the RAM locations refer to an AT-mega128. The memory addresses used are not displayed in a linear scale.

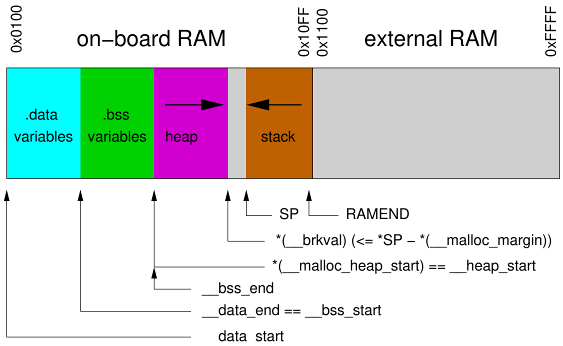

Figure 1 RAM map of a device with internal RAM

On a simple device like a microcontroller it is a challenge to implement a dynamic memory allocator that is simple enough so the code size requirements will remain low, yet powerful enough to avoid unnecessary memory fragmentation and to get it all done with reasonably few CPU cycles. Microcontrollers are often low on space and also run at much lower speeds than the typical PC these days.

The memory allocator implemented in AVR-LibC tries to cope with all of these constraints, and offers some tuning options that can be used if there are more resources available than in the default configuration.

3.2 Internal vs. external RAM

Obviously, the constraints are much harder to satisfy in the default configuration where only internal RAM is available. Extreme care must be taken to avoid a stack-heap collision, both by making sure functions aren't nesting too deeply, and don't require too much stack space for local variables, as well as by being cautious with allocating too much dynamic memory.

If external RAM is available, it is strongly recommended to move the heap into the external RAM, regardless of whether or not the variables from the .data and .bss sections are also going to be located there. The stack should always be kept in internal RAM. Some devices even require this, and in general, internal RAM can be accessed faster since no extra wait states are required. When using dynamic memory allocation and stack and heap are separated in distinct memory areas, this is the safest way to avoid a stack-heap collision.

3.3 Tunables for malloc()

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

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

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

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

```
avr-gcc ... -Wl,--section-start,.data=0x801100,--defsym=__heap_end=0x80ffff ...
```

Note

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

The ld (linker) user manual states that using -Tdata=<x> is equivalent to using –section-start,.data=<x>. However, you have to use –section-start as above because the GCC frontend also sets the -Tdata option for all MCU types where the SRAM doesn't start at 0x800060. Thus, the linker is being faced with two -Tdata options. Sarting with binutils 2.16, the linker changed the preference, and picks the "wrong" option in this situation.

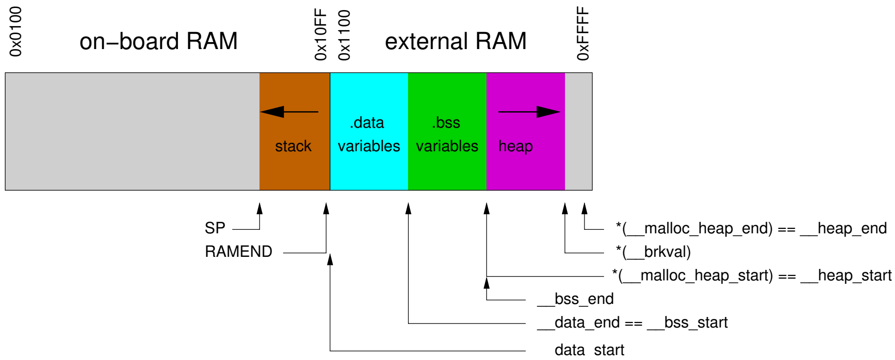

Figure 2 Internal RAM: stack only, external RAM: variables and heap

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

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

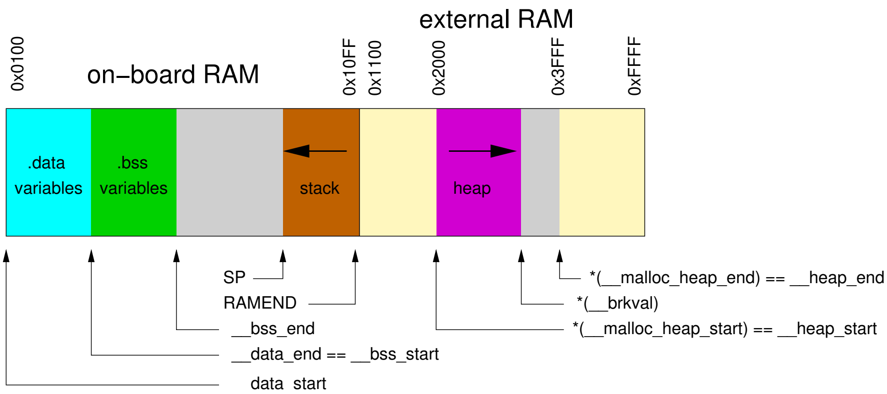

Figure 3 Internal RAM: variables and stack, external RAM: heap

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

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

3.4 Implementation details

Dynamic memory allocation requests will be returned with a two-byte header prepended that records the size of the allocation. This is later used by free(). The returned address points just beyond that header. Thus, if the application accidentally writes before the returned memory region, the internal consistency of the memory allocator is compromised.

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

The freelist itself is not maintained as a separate data structure, but rather by modifying the contents of the freed memory to contain pointers chaining the pieces together. That way, no additional memory is reqired to maintain this list except for a variable that keeps track of the lowest memory segment available for reallocation. Since both, a chain pointer and the size of the chunk need to be recorded in each chunk, the minimum chunk size on the freelist is four bytes.

When allocating memory, first the freelist is walked to see if it could satisfy the request. If there's a chunk available on the freelist that will fit the request exactly, it will be taken, disconnected from the freelist, and returned to the caller. If no exact match could be found, the closest match that would just satisfy the request will be used. The chunk will normally be split up into one to be returned to the caller, and another (smaller) one that will remain on the freelist. In case this chunk was only up to two bytes larger than the request, the request will simply be altered internally to also account for these additional bytes since no separate freelist entry could be split off in that case.

If nothing could be found on the freelist, heap extension is attempted. This is where `__malloc_margin` will be considered if the heap is operating below the stack, or where `__malloc_heap_end` will be verified otherwise.

If the remaining memory is insufficient to satisfy the request, `NULL` will eventually be returned to the caller.

When calling free(), a new freelist entry will be prepared. An attempt is then made to aggregate the new entry with possible adjacent entries, yielding a single larger entry available for further allocations. That way, the potential for heap fragmentation is hopefully reduced. When deallocating the topmost chunk of memory, the size of the heap is reduced.

A call to realloc() first determines whether the operation is about to grow or shrink the current allocation. When shrinking, the case is easy: the existing chunk is split, and the tail of the region that is no longer to be used is passed to the standard free() function for insertion into the freelist. Checks are first made whether the tail chunk is large enough to hold a chunk of its own at all, otherwise realloc() will simply do nothing, and return the original region.

When growing the region, it is first checked whether the existing allocation can be extended in-place. If so, this is done, and the original pointer is returned without copying any data contents. As a side-effect, this check will also record the size of the largest chunk on the freelist.

If the region cannot be extended in-place, but the old chunk is at the top of heap, and the above freelist walk did not reveal a large enough chunk on the freelist to satisfy the new request, an attempt is made to quickly extend this topmost chunk (and thus the heap), so no need arises to copy over the existing data. If there's no more space available in the heap (same check is done as in malloc()), the entire request will fail.

Otherwise, malloc() will be called with the new request size, the existing data will be copied over, and free() will be called on the old region.

4 Memory Sections

Section are used to organize code and data of a program on the binary level.

The (compiler-generated) assembly code assigns code, data and other entities like debug information to so called input sections. These sections serve as input to the linker, which bundles similar sections together to output sections like `.text` and `.data` according to rules defined in the linker description file.

The final ELF binary is then used by programming tools like `avrdude`, simulators, debuggers and other programs, for example programs from the GNU Binutils family like `avr-size`, `avr-objdump` and `avr-readelf`.

Sections may have extra properties like section alignment, section flags, section type and rules to locate them or to assign them to memory regions.

4.1 Concepts

4.1.1 Named Sections

Named sections are sections that can be referred to by their name. The name and other properties can be provided with the `.section` directive like in

```
.section name, "flags", @type
```

or with the `.pushsection` directive, which directs the assembler to assemble the following code into the named section.

An example of a section that is not referred to by its name is the COMMON section. In order to put an object in that section, special directives like `.comm name,size` or `.lcomm name,size` have to be used.

Directives like `.text` are basically the same like `.section .text`, where the assembler assumes appropriate section flags and type; same for directives `.data` and `.bss`.

4.1.1.1 Section Flags

The *section flags* can be specified with the `.section` and `.pushsection` directives, see section type for an example. Section flags of output sections can be specified in the linker description file, and the linker implements heuristics to determine the section flags of output sections from the various input section that go into it.

Table 1 Section Flags

Flag	Meaning
`a`	The section will be **a**llocated, i.e. it occupies space on the target hardware
`w`	The section contains data that can be **w**ritten at run-time. Sections that only contain read-only entities don't have the `w` flag set
`x`	The section contains e**x**ecutable code, though the section may also contain non-executable objects
`M`	A **m**ergeable section
`S`	A **s**tring section
`G`	A section **g**roup, like used with `comdat` objects

The last three flags are listed for completeness. They are used by the compiler, for example for header-only C++ modules and to ensure that multiplle instanciations of the same template in different compilaton units does occur at most once in the executable file.

4.1.1.2 Section Type

The *section type* can be specified with the `.section` and `.pushsection` directives, like in

```
.section .text.myfunc,"ax",@progbits
.pushsection ".data.myvar", "a", "progbits"
```

On ELF level, the section type is stored in the section header like `Elf32_Shdr.sh_type = SHT_PROGBITS`.

Table 2 Section Types

Type	Meaning
`@progbits`	The section contains data that will be loaded to the target, like objects in the `.text` and `.data` sections.
`@nobits`	The section does not contain data that needs to be transferred to the target device, like data in the `.bss` and `.noinit` sections. The section still occupies space on the target.
`@note`	The section is a note, like for example the .note.gnu.avr.deviceinfo section.

4.1.1.3 Section Alignment The *alignment* of a section is the maximum over the alignments of the objects in the section.

4.1.1.4 Subsections *Subsections* are compartments of named sections and are introduced with the `.subsection` directive. Subsections are located in order of increasing index in their input section. The default subsection after switching to a new section is subsection 0.

Note

> A common misconception is that a section like `.text.module.func` were a subsection of `.text.module`. This is not the case. These two sections are independent, and there is no subset relation. The sections may have different flags and type, and they may be assigned to different output sections.

4.1.2 Orphan Sections

Orphan sections are sections that are not mentioned in the linker description file. When an input section is orphan, then the GNU linker implicitly generates an output section of the same name. The linker implements various heuristics to determine sections flags, section type and location of orphaned sections. One use of orphan sections is to locate code to a fixed address.

Like for any other output section, the start address can be specified by means of linking with `-Wl,--section-start,`*secname=*

4.1.3 LMA: Load Memory Address

The LMA of an object is the address where a loader like `avrdude` puts the object when the binary is being uploaded to the target device.

4.1.4 VMA: Virtual Memory Address

The VMA is the address of an object as used by the running program.

VMA and LMA may be different: Suppose a small ATmega8 program with executable code that extends from byte address 0x0 to 0x20f, and one variable `my_var` in static strorage. The default linker script puts the content of the `.data` output section after the `.text` output section and into the `text` segment. The startup code then copies `my_data` from its LMA location beginning at 0x210 to its VMA location beginning at 0x800060, because C/C++ requires that all data in static storage must have been initialized when `main` is entered.

The internal SRAM of ATmega8 starts at RAM address 0x60, which is offset by 0x800000 in order to linearize the address space (VMA 0x60 is a flash address). The AVR program only ever uses the lower 16 bits of VMAs in static storage so that the offset of 0x800000 is masked out. But code like "`LDI r24,hh8(my_data)`" actually sets R24 to 0x80 and reveals that `my_data` is an object located in RAM.

4.2 The Linker Script: Building Blocks

The linker description file is the central hub to channel functions and static storage objects of a program to the various memory spaces and address ranges of a device.

4.2.1 Input Sections and Output Sections

Input sections are sections that are inputs to the linker. Functions and static variables but also additional notes and debug information are assigned to different input sections by means of assembler directives like `.section` or `.text`. The linker takes all these sections and assigns them to output sections as specified in the linker script.

Output sections are defind in the linker description file. Contrary to the unlimited number of input sections a program can come up with, there is only a handfull of output sections like `.text` and `.data`, that roughly correspond to the memory spaces of the target device.

One step in the final link is to *locate* the sections, that is the linker/locator determines at which memory location to put the output sections, and how to arrange the many input sections within their assigned output section. *Locating* means that the linker assigns Load Memory Addresses — addresses as used by a loader like `avrdude` — and Virtual Memory Addresses, which are the addresses as used by the running program.

While it is possible to directly assign LMAs and VMAs to output sections in the linker script, the default linker scripts provided by Binutils assign *memory regions* (aka. *memory segments*) to the output sections. This has some advantages like a linker script that is easier to maintain. An output sections can be assigned to more than one memory region. For example, non-zero data in static storage (`.data`) goes to

1. the `data` region (VMA), because such variables occupy RAM which has to be allocated
2. the `text` region (LMA), because the initializers for such data has to be kept in some non-volatile memory (program ROM), so that the startup code can initialize that data so that the variables have their expected initial values when `main()` is entered.

The `SECTIONS{}` portion of a linker script models the input and output section, and it assignes the output section to the memory regions defined in the `MEMORY{}` part.

4.2.2 Memory Regions

The *memory regions* defined in the default linker script model and correspond to the different kinds of memories of a device.

Table 3 Memory Regions of the Default Linker Script

Region	Virtual Address[1]	Flags	Purpose
`text`	0[2]	`rx`	Executable code, vector table, data in PROGMEM, `__flash` and `__memx`, startup code, linker stubs, initializers for `.data`
`data`	`0x800000`[2]	`rw`	Data in static storage
`rodata`[3]	`0xa00000`[2]	`r`	Read-only data in static storage
`eeprom`	`0x810000`	`rw`	EEPROM data
`fuse`	`0x820000`	`rw`	Fuse bytes
`lock`	`0x830000`	`rw`	Lock bytes
`signature`	`0x840000`	`rw`	Device signature
`user_signatures`	`0x850000`	`rw`	User signature

Notes

1. The VMAs for regions other than `text` are offset in order to linearize the non-linear memory address space

of the AVR Harvard architecture. The target code only ever uses the lower 16 bits of the VMA to access objects in non-`text` regions.

2. The addresses for regions `text`, `data` and rodata are actually defined as symbols like `__TEXT_REGION_ORIGIN__`, so that they can be adjusted by means of, say `-Wl,--defsym,__DATA_REGION_ORIGIN__=0x800060`. Same applies for the lengths of all the regions, which is `__`*NAME*`_REGION_LENGTH__` for region *name*.
3. The `rodata` region is only present in the `avrxmega2_flmap` and `avrxmega4_flmap` emulations, which is the case for Binutils since v2.42 for the AVR64 and AVR128 devices without `-mrodata-in-ram`.

4.3 Output Sections of the Default Linker Script

This section describes the various output sections defined in the default linker description files.

Table 4 Output Sections and Memory Regions

Output Section	Purpose	Memory Region	
		LMA	VMA
`.text`	Executable code, data in progmem	text	text
`.data`	Non-zero data in static storage	text	data
`.bss`	Zero data in static storage	—	data
`.noinit`	Non-initialized data in static storage	—	data
`.rodata`[1]	Read-only data in static storage	text	LMA + offset[3]
`.rodata`[2]	Read-only data in static storage	0x8000 ∗ `__flmap`[4]	rodata
`.eeprom`	Data in EEPROM		eeprom
`.fuse`	Fuse bytes		fuse
`.lock`	Lock bytes	Note[5]	lock
`.signature`	Signature bytes		signature
	User signature bytes		user_signatures

Notes

1. On `avrxmega3` and `avrtiny` devices.
2. On `AVR64` and `AVR128` devices without `-mrodata-in-ram`.
3. With an offset `__RODATA_PM_OFFSET__` of 0x4000 or 0x8000 depending on the device.
4. The value of symbol `__flmap` defaults to the last 32 KiB block of program memory, see the GCC `v14 release notes`.
5. The LMA actually equals the VMA, but is unused. The flash loader like avrdude knows where to put the data,

4.3.1 The .text Output Section

The `.text` output section contains the actual machine instructions which make up the program, but also additional code like jump tables and lookup tables placed in program memory with the PROGMEM attribute.

The `.text` output section contains the input sections described below. Input sections that are not used by the tools are omitted. A ∗ wildcard stands for any sequence of characters, including empty ones, that are valid in a section name.

.vectors The .vectors sections contains the interrupt vector table which consists of jumps to weakly defined labels: To __init for the first entry at index 0, and to __vector_*N* for the entry at index $N \geq 1$. The default value for __vector_*N* is __bad_interrupt, which jumps to weakly defined __vector_default, which jumps to __vectors, which is the start of the .vectors section.

Implementing an interrupt service ruotine (ISR) is performed with the help of the ISR macro in C/C++ code.

.progmem.data

.progmem.data.*

.progmem.gcc.* This section is used for read-only data declared with attribute PROGMEM, and for data in address-space __flash.

The compiler assumes that the .progmem sectons are located in the lower 64 KiB of program memory. When it does not fit in the lower 64 KiB block, then the program reads garbage except pgm_read_*_far is used. In that case however, code can be located in the .progmemx section which does not require to be located in the lower program memory.

.trampolines Linker stubs for indirect jumps and calls on devices with more than 128 KiB of program memory. This section must be located in the same 128 KiB block like the interrupt vector table. For some background on linker stubs, see the GCC documentation on EIND.

.text

.text.* Executable code. This is where almost all of the executable code of an application will go.

.ctors

.dtors Tables with addresses of static constructors and destructors, like C++ static constructors and functions declared with attribute constructor.

The .init*N* Sections These sections are used to hold the startup code from reset up through the start of main().

The .init*N* sections are executed in order from 0 to 9: The code from one init section falls through to the next higher init section. This is the reason for why code in these sections must be naked (more precisely, it must not contain return instructions), and why code in these sections must never be called explicitly.

When several modules put code in the same init section, the order of execuation is not specified.

Table 5 The .init*N* Sections

Section	Performs	Hosted By	Symbol[1]
.init0	Weakly defines the __init label which is the jump target of the first vector in the interrupt vector table. When the user defines the __init() function, it will be jumped to instead.	AVR-LibC[2]	
.init1	Unused	—	
.init2	• Clears __zero_reg__ • Initializes the stack pointer to the value of weak symbol __stack, which has a default value of RAMEND as defined in avr/io.h • Initializes EIND to hh8(pm(__vectors)) on devices that have it • Initializes RAMPX, RAMPY, RAMPZ and RAMPD on devices that have all of them	AVR-LibC	
.init3	Initializes the NVMCTRLB.FLMAP bit-field on devices that have it, except when -mrodata-in-ram is specified	AVR-LibC	__do_flmap_init

Section	Performs	Hosted By	Symbol[1]
.init4	Initializes data in static storage: Initializes .data and clears .bss	libgcc	__do_copy_data __do_clear_bss
.init5	Unused	—	
.init6	Run static C++ constructors and functions defined with __attribute__((constructor)).	libgcc	__do_global_ctors
.init7	Unused	—	
.init8	Unused	—	
.init9	Calls main and then jumps to exit	AVR-LibC	

Notes

1. Code in the .init3, .init4 and .init6 sections is optional; it will only be present when there is something to do. This will be tracked by the compiler — or has to be tracked by the assembly programmer — which pulls in the code from the respective library by means of the mentioned symbols, e.g. by linking with -Wl,-u,__do_flmap_init or by means of
```
.global __do_copy_data
```
Conversely, when the respective code is not desired for some reason, the symbol can be satisfied by defining it with, say, -Wl,--defsym,__do_copy_data=0 so that the code is not pulled in any more.
2. The code is provided by gcrt1.S.

The .fini*N* Sections Shutdown code. These sections are used to hold the exit code executed after return from main() or a call to exit().

The .fini*N* sections are executed in descending order from 9 to 0 in a fallthrough manner.

Table 6 The .fini*N* Sections

Section	Performs	Hosted By	Symbol
.fini9	Defines _exit and weakly defines the exit label	libgcc	
.fini8	Run functions registered with atexit()	AVR-LibC	
.fini7	Unused	—	
.fini6	Run static C++ destructors and functions defined with __attribute__((destructor))	libgcc	__do_global_dtors
.fini5...1	Unused	—	
.fini0	Globally disables interrupts and enters an infinite loop to label __stop_program	libgcc	

It is unlikely that ordinary code uses the fini sections. When there are no static destructors and atexit() is not used, then the respective code is not pulled in form the libraries, and the fini code just consumes four bytes: a CLI and a RJMP to itself. Common use cases of fini code is when running the GCC test suite where it reduces fallout, and in simulators to determine (un)orderly termination of a simulated program.

.progmemx.* Read-only data in program memory without the requirement that it must reside in the lower 64 KiB. The compiler uses this section for data in the named address-space __memx. Data can be accessed with pgm_read_*_far when it is not in a named address-space:
```
#include <avr/pgmspace.h>

const __memx int array1[] = { 1, 4, 9, 16, 25, 36 };

PROGMEM_FAR
const int array2[] = { 2, 3, 5, 7, 11, 13, 17 };

int add (uint8_t id1, uint8_t id2)
{
    uint_farptr_t p_array2 = pgm_get_far_address (array2);
```

```
        int val2 = pgm_read_int_far (p_array2 + sizeof(int) * id2);

        return val2 + array1[id1];
    }
```

.jumptables* Used to place jump tables in some cases.

4.3.2 The .data Output Section

This section contains data in static storage which has an initializer that is not all zeroes. This includes the following input sections:

.data* Read-write data

.rodata* Read-only data. These input sections are only included on devices that host read-only data in RAM.

It is possible to tell the linker the SRAM address of the beginning of the `.data` section. This is accomplished by linking with

```
avr-gcc ... -Tdata addr -Wl,--defsym,__DATA_REGION_START__=addr
```

Note that `addr` must be offset by adding 0x800000 the to real SRAM address so that the linker knows that the address is in the SRAM memory segment. Thus, if you want the `.data` section to start at `0x1100`, pass `0x801100` as the address to the linker.

Note

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

4.3.3 The .bss Output Section

Data in static storage that will be zeroed by the startup code. This are data objects without explicit initializer, and data objects with initializers that are all zeroes.

Input sections are `.bss*` and `COMMON`. Common symbols are defined with directives `.comm` or `.lcomm`.

4.3.4 The .noinit Output Section

Data objects in static storage that should not be initialized by the startup code. As the C/C++ standard requires that *all* data in static storage is initialized — which includes data without explicit initializer, which will be initialized to all zeroes — such objects have to be put into section `.noinit` by hand:

```
__attribute__ ((section (".noinit")))
int foo;
```

The only input section in this output section is `.noinit`. Only data without initializer can be put in this section.

4.3.5 The .rodata Output Section

This section contains read-only data in static storage from `.rodata*` input sections. This output section is only present for devices where read-only data remains in program memory, which are the devices where (parts of) the program memory are visible in the RAM address space. This is currently the case for the emulations `avrtiny`, `avrxmega3`, `avrxmega2_flmap` and `avrxmega4_flmap`.

4.3.6 The .eeprom Output Section

This is where EEPROM variables are stored, for example variables declared with the `EEMEM` attribute. The only input section (pattern) is `.eeprom*`.

4.3.7 The .fuse, .lock and .signature Output Sections

These sections contain fuse bytes, lock bytes and device signature bytes, respectively. The respective input section patterns are `.fuse*` `.lock*` and `.signature*`.

4.3.8 The .note.gnu.avr.deviceinfo Section

This section is actually *not mentioned* in the default linker script, which means it is an orphan section and hence the respective output section is implicit.

The startup code from AVR-LibC puts device information in that section to be picked up by simulators or tools like avr-size, avr-objdump, avr-readelf, etc,

The section is contained in the ELF file but not loaded onto the target. Source of the device specific information are the device header file and compiler builtin macros. The layout conforms to the standard `ELF note section` layout and is laid out as follows.

```
#include <elf.h>

typedef struct
{
    Elf32_Word n_namesz;     /* AVR_NOTE_NAME_LEN */
    Elf32_Word n_descsz;     /* size of avr_desc */
    Elf32_Word n_type;       /* 1 - the only AVR note type */
} Elf32_Nhdr;

#define AVR_NOTE_NAME_LEN 4

struct note_gnu_avr_deviceinfo
{
    Elf32_Nhdr nhdr;
    char note_name[AVR_NOTE_NAME_LEN]; /* = "AVR\0" */

    struct
    {
        Elf32_Word flash_start;
        Elf32_Word flash_size;
        Elf32_Word sram_start;
        Elf32_Word sram_size;
        Elf32_Word eeprom_start;
        Elf32_Word eeprom_size;
        Elf32_Word offset_table_size;
        /* Offset table containing byte offsets into
           string table that immediately follows it.
           index 0: Device name byte offset */
        Elf32_Off offset_table[1];
        /* Standard ELF string table.
           index 0 : NULL
           index 1 : Device name
           index 2 : NULL */
        char strtab[2 + strlen(__AVR_DEVICE_NAME__)];
    } avr_desc;
};
```

The contents of this section can be displayed with

- `avr-objdump -P avr-deviceinfo` *`file`*, which is supported since Binutils v2.43.
- `avr-readelf -n` *`file`*, which displays all notes.

4.4 Symbols in the Default Linker Script

Most of the symbols like `main` are defined in the code of the application, but some symbols are defined in the default linker script:

__*name*_REGION_ORIGIN__ Describes the physical properties of memory region *name*, where *name* is one of `TEXT` or `DATA`. The address is a VMA and offset at explained above.
The linker script only supplies a default for the symbol values when they have not been defined by other means, like for example in the startup code or by `--defsym`. For example, to let the code start at address `0x100`, one can link with

```
avr-gcc ... -Ttext=0x100 -Wl,--defsym,__TEXT_REGION_ORIGIN__=0x100
```

__*name*_REGION_LENGTH__ Describes the physical properties of memory region *name*, where *name* is one of: `TEXT`, `DATA`, `EEPROM`, `LOCK`, `FUSE`, `SIGNATURE` or `USER_SIGNATURE`.
Only a default is supplied when the symbol is not yet defined by other means. Most of these symbols are weakly defined in the startup code.

__data_start

__data_end Start and (one past the) end VMA address of the `.data` section in RAM.

__data_load_start

__data_load_end Start and (one past the) end LMA address of the `.data` section initializers located in program memory. Used together with the VMA addresses above by the startup code to copy data initializers from program memory to RAM.

__bss_start

__bss_end Start and (one past the) end VMA address of the `.bss` section. The startup code clears this part of the RAM.

__rodata_start

__rodata_end

__rodata_load_start

__rodata_load_end Start and (one past the) end VMA resp. LMA address of the `.rodata` output section. These symbols are only defined when `.rodata` is not output to the `text` region, which is the case for emulations `avrxmega2_flmap` and `avrxmega4_flmap`.

__heap_start One past the last object located in static storage. Immediately follows the `.noinit` section (which immediately follows `.bss`, which immediately follows `.data`). Used by malloc() and friends.

Code that computes a checksum over all relevant code and data in program memory has to consider:

- The range from the beginning of the `.text` section (address 0x0 in the default layout) up to `__data_load_start`.
- For emulations that have the `rodata` memory region, the range from `__rodata_load_start` to `__rodata_load_end` has also to be taken into account.

4.5 Output Sections and Code Size

The `avr-size` program (part of Binutils), coming from a Unix background, doesn't account for the `.data` initialization space added to the `.text` section, so in order to know how much flash the final program will consume, one needs to add the values for both, `.text` and `.data` (but not `.bss`), while the amount of pre-allocated SRAM is the sum of `.data` and `.bss`.

Memory usage and free memory can also be displayed with

```
avr-objdump -P mem-usage code.elf
```

4.6 Using Sections

4.6.1 In C/C++ Code

The following example shows how to read and reset the `MCUCR` special function register on ATmega328. This SFR holds to reset source like "watchdog reset" or "external reset", and should be read early, prior to the initialization of RAM and execution of static constructors which may take some time. This means the code has to be placed prior to `.init4` which initializes static storage, but after `.init2` which initializes `__zero_reg__`. As the code runs prior to the initialization of static storage, variable `mcucr` must be placed in section `.noinit` so that it won't be overridden by that part of the startup code:

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

__attribute__((section(".noinit")))
uint8_t mcucr;

__attribute__((used, unused, naked, section(".init3")))
static void read_MCUCR (void)
{
    mcucr = MCUCR;
    MCUCR = 0;
}
```

- The `used` attribute tells the compiler that the function is used although it is never called.
- The `unused` attribute tells the compiler that it is fine that the function is unused, and silences respective diagnostics about the seemingly unused functions.
- The `naked` attribute is required because the code is located in an init section. The function *must not have a `RET` statement* because the function is never called. According to the GCC documentation, the only code supported in naked functions is inline assembly, but the code above is simple enough so that GCC can deal with it.

4.6.2 In Assembly Code

Example:

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

.section .init3,"ax",@progbits
    lds     r0, MCUCR

.pushsection .noinit,"a",@nobits
mcucr:
    .type   mcucr, @object
    .size   mcucr, 1
    .space  1
.popsection                     ; Proceed with .init3

    sts     mcucr, r0
    sts     MCUCR, __zero_reg__ ; Initialized in .init2

.text
    .global main
    .type   main, @function
    lds     r24,    mcucr
    clr     r25
    rjmp    putchar
    .size main, .-main
```

- The `"ax"` flags tells that the sections is **a**llocatable (consumes space on the target hardware) and is e**x**ecutable.
- The `@progbits` type tells that the section contains bits that have to be uploaded to the target hardware.

For more detais, see the see the gas user manual on the `.section` directive.

5 Data in Program Space

5.1 Introduction

So you have some constant data and you're running out of room to store it? Many AVRs have limited amount of RAM in which to store data, but may have more Flash space available. The AVR is a Harvard architecture processor, where Flash is used for the program, RAM is used for data, and they each have separate address spaces. It is a challenge to get constant data to be stored in the Program Space, and to retrieve that data to use it in the AVR application.

The problem is exacerbated by the fact that the C Language was not designed for Harvard architectures, it was designed for Von Neumann architectures where code and data exist in the same address space. This means that any compiler for a Harvard architecture processor, like the AVR, has to use other means to operate with separate address spaces.

Some compilers use non-standard C language keywords, or they extend the standard syntax in ways that are non-standard. The AVR toolset takes a different approach.

GCC has a special keyword, `__attribute__` that is used to attach different attributes to things such as function declarations, variables, and types. This keyword is followed by an attribute specification in double parentheses. In AVR GCC, there is a special attribute called `progmem`. This attribute is use on data declarations, and tells the compiler to place the data in the Program Memory (Flash).

AVR-LibC provides a simple macro `PROGMEM` that is defined as the attribute syntax of GCC with the `progmem` attribute. This macro was created as a convenience to the end user, as we will see below. The `PROGMEM` macro is defined in the `<avr/pgmspace.h>` system header file.

It is difficult to modify GCC to create new extensions to the C language syntax, so instead, AVR-LibC has created macros to retrieve the data from the Program Space. These macros are also found in the `<avr/pgmspace.h>` system header file.

5.2 A Note On const

Many users bring up the idea of using C's keyword `const` as a means of declaring data to be in Program Space. Doing this would be an abuse of the intended meaning of the `const` keyword.

`const` is used to tell the compiler that the data is to be "read-only". It is used to help make it easier for the compiler to make certain transformations, or to help the compiler check for incorrect usage of those variables.

For example, the const keyword is commonly used in many functions as a modifier on the parameter type. This tells the compiler that the function will only use the parameter as read-only and will not modify the contents of the parameter variable.

`const` was intended for uses such as this, not as a means to identify where the data should be stored. If it were used as a means to define data storage, then it loses its correct meaning (changes its semantics) in other situations such as in the function parameter example.

5.3 Storing and Retrieving Data in the Program Space

Let's say you have some global data:

```
unsigned char mydata[11][10] =
{
    {0x00,0x01,0x02,0x03,0x04,0x05,0x06,0x07,0x08,0x09},
    {0x0A,0x0B,0x0C,0x0D,0x0E,0x0F,0x10,0x11,0x12,0x13},
    {0x14,0x15,0x16,0x17,0x18,0x19,0x1A,0x1B,0x1C,0x1D},
    {0x1E,0x1F,0x20,0x21,0x22,0x23,0x24,0x25,0x26,0x27},
    {0x28,0x29,0x2A,0x2B,0x2C,0x2D,0x2E,0x2F,0x30,0x31},
    {0x32,0x33,0x34,0x35,0x36,0x37,0x38,0x39,0x3A,0x3B},
    {0x3C,0x3D,0x3E,0x3F,0x40,0x41,0x42,0x43,0x44,0x45},
    {0x46,0x47,0x48,0x49,0x4A,0x4B,0x4C,0x4D,0x4E,0x4F},
    {0x50,0x51,0x52,0x53,0x54,0x55,0x56,0x57,0x58,0x59},
    {0x5A,0x5B,0x5C,0x5D,0x5E,0x5F,0x60,0x61,0x62,0x63},
    {0x64,0x65,0x66,0x67,0x68,0x69,0x6A,0x6B,0x6C,0x6D}
};
```

and later in your code you access this data in a function and store a single byte into a variable like so:

```
byte = mydata[i][j];
```

Now you want to store your data in Program Memory. Use the `PROGMEM` macro found in <avr/pgmspace.h> and put it after the declaration of the variable, but before the initializer, like so:

```
#include <avr/pgmspace.h>
.
.
.
const unsigned char mydata[11][10] PROGMEM =
{
    {0x00,0x01,0x02,0x03,0x04,0x05,0x06,0x07,0x08,0x09},
    {0x0A,0x0B,0x0C,0x0D,0x0E,0x0F,0x10,0x11,0x12,0x13},
    {0x14,0x15,0x16,0x17,0x18,0x19,0x1A,0x1B,0x1C,0x1D},
    {0x1E,0x1F,0x20,0x21,0x22,0x23,0x24,0x25,0x26,0x27},
    {0x28,0x29,0x2A,0x2B,0x2C,0x2D,0x2E,0x2F,0x30,0x31},
    {0x32,0x33,0x34,0x35,0x36,0x37,0x38,0x39,0x3A,0x3B},
    {0x3C,0x3D,0x3E,0x3F,0x40,0x41,0x42,0x43,0x44,0x45},
    {0x46,0x47,0x48,0x49,0x4A,0x4B,0x4C,0x4D,0x4E,0x4F},
    {0x50,0x51,0x52,0x53,0x54,0x55,0x56,0x57,0x58,0x59},
    {0x5A,0x5B,0x5C,0x5D,0x5E,0x5F,0x60,0x61,0x62,0x63},
    {0x64,0x65,0x66,0x67,0x68,0x69,0x6A,0x6B,0x6C,0x6D}
};
```

That's it! Now your data is in the Program Space. You can compile, link, and check the map file to verify that `mydata` is placed in the correct section.

Now that your data resides in the Program Space, your code to access (read) the data will no longer work. The code that gets generated will retrieve the data that is located at the address of the `mydata` array, plus offsets indexed by the `i` and `j` variables. However, the final address that is calculated where to the retrieve the data points to the Data Space! Not the Program Space where the data is actually located. It is likely that you will be retrieving some garbage. The problem is that AVR GCC does not intrinsically know that the data resides in the Program Space.

The solution is fairly simple. The "rule of thumb" for accessing data stored in the Program Space is to access the data as you normally would (as if the variable is stored in Data Space), like so:

```
byte = mydata[i][j];
```

then take the address of the data:

```
byte = &(mydata[i][j]);
```

then use the appropriate `pgm_read_*` macro, and the address of your data becomes the parameter to that macro:

```
byte = pgm_read_byte(&(mydata[i][j]));
```

The `pgm_read_*` macros take an address that points to the Program Space, and retrieves the data that is stored at that address. This is why you take the address of the offset into the array. This address becomes the parameter to the macro so it can generate the correct code to retrieve the data from the Program Space. There are different `pgm_read_*` macros to read different sizes of data at the address given.

5.4 Storing and Retrieving Strings in the Program Space

Now that you can successfully store and retrieve simple data from Program Space you want to store and retrive strings from Program Space. And specifically you want to store and array of strings to Program Space. So you start off with your array, like so:

```
const char* const string_table[] =
{
    "String 1",
    "String 2",
    "String 3",
    "String 4",
    "String 5"
};
```

and then you add your PROGMEM macro to the end of the declaration:

```
const char* const string_table[] PROGMEM =
{
    "String 1",
    "String 2",
    "String 3",
    "String 4",
    "String 5"
};
```

Right? WRONG!

Unfortunately, with GCC attributes, they affect only the declaration that they are attached to. So in this case, we successfully put the `string_table` variable, the array itself, in the Program Space. This DOES NOT put the actual strings themselves into Program Space. At this point, the strings are still in the Data Space, which is probably not what you want.

In order to put the strings in Program Space, you have to have explicit declarations for each string, and put each string in Program Space:

```
const char string_1[] PROGMEM = "String 1";
const char string_2[] PROGMEM = "String 2";
const char string_3[] PROGMEM = "String 3";
const char string_4[] PROGMEM = "String 4";
const char string_5[] PROGMEM = "String 5";
```

Then use the new symbols in your table, like so:

```
const char* const string_table[] PROGMEM =
{
    string_1,
    string_2,
    string_3,
    string_4,
    string_5
};
```

Now this has the effect of putting `string_table` in Program Space, where `string_table` is an array of pointers to characters (strings), where each pointer is a pointer to the Program Space, where each string is also stored.

Retrieving the strings are a different matter. You probably don't want to pull the string out of Program Space, byte by byte, using the `pgm_read_byte()` macro. There are other functions declared in the <avr/pgmspace.h> header file that work with strings that are stored in the Program Space.

For example if you want to copy the string from Program Space to a buffer in RAM (like an automatic variable inside a function, that is allocated on the stack), you can do this:

```
void foo(void)
{
    char buffer[10];

    for (uint8_t i = 0; i < 5; i++)
    {
        strcpy_P(buffer, (const char*) pgm_read_ptr(&(string_table[i])));

        // Display buffer on LCD.
    }
    return;
}
```

Here, the string_table array is stored in Program Space, so we access it normally, as if were stored in Data Space, then take the address of the location we want to access, and use the address as a parameter to pgm↵_read_ptr. We use the pgm_read_ptr macro to read the string pointer out of the string_table array. Remember that a pointer is 16-bits, or word size. The pgm_read_ptr macro will return a void*. This pointer is an address in Program Space pointing to the string that we want to copy. This pointer is then used as a parameter to the function strcpy_P. The function strcpy_P is just like the regular strcpy function, except that it copies a string from Program Space (the second parameter) to a buffer in the Data Space (the first parameter).

There are many string functions available that work with strings located in Program Space. All of these special string functions have a suffix of _P in the function name, and are declared in the <avr/pgmspace.h> header file.

5.5 Caveats

The macros and functions used to retrieve data from the Program Space have to generate some extra code in order to actually load the data from the Program Space. This incurs some extra overhead in terms of code space (extra opcodes) and execution time. Usually, both the space and time overhead is minimal compared to the space savings of putting data in Program Space. But you should be aware of this so you can minimize the number of calls within a single function that gets the same piece of data from Program Space. It is always instructive to look at the resulting disassembly from the compiler.

6 AVR-LibC and Assembler Programs

- Introduction
- Invoking the Compiler
- Example Program
- Assembler Directives
  - Sections
  - Symbols
  - Data and Alignment
- Operand Modifiers

6.1 Introduction

There might be several reasons to write code for AVR microcontrollers using plain assembler source code. Among them are:

- Code for devices that do not have RAM and are thus not supported by the C compiler.
- Code for very time-critical applications.
- Special tweaks that cannot be done in C.

Usually, all but the first could probably be done easily using the inline assembler facility of the compiler.

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

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

6.2 Invoking the Compiler

For the purpose described in this document, the assembler and linker are usually not invoked manually, but rather using the C compiler frontend (`avr-gcc`) that in turn will call the assembler and linker as required.

This approach has the following advantages:

- There is basically only one program to be called directly, `avr-gcc`, regardless of the actual source language used.
- The invokation of the C preprocessor will be automatic, and will include the appropriate options to locate required include files in the filesystem.
- The invokation of the linker will be automatic, and will include the appropriate options to locate additional libraries as well as the application start-up code (`crt`*XXX*`.o`) and linker script.

Note that the invokation of the C preprocessor will be automatic when the filename provided for the assembler file ends in `.S` (the capital letter "s"). This would even apply to operating systems that use case-insensitive filesystems since the actual decision is made based on the case of the filename suffix given on the command-line, not based on the actual filename from the file system.

As an alternative to using `.S`, the suffix `.sx` is recognized for this purpose (starting with GCC v4.3). This is primarily meant to be compatible with other compiler environments that have been providing this variant before in order to cope with operating systems where filenames are case-insensitive (and, with some versions of `make` that could not distinguish between `.s` and `.S` on such systems).

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

6.3 Example Program

The following annotated example features a simple 100 kHz square wave generator using an AT90S1200 clocked with a 10.7 MHz crystal. Pin PD6 will be used for the square wave output.

```
#include <avr/io.h>             // Note [1]

work    =   16                  // Note [2]
tmp     =   17

inttmp  =   19
intsav  =   0

SQUARE  =   PD6                 // Note [3]

#define IO(x) _SFR_IO_ADDR(x)

// Note [4]:
// 100 kHz => 200000 edges/s
tmconst = 10700000 / 200000

// # clocks in ISR until TCNT0 is set
fuzz = 8

.text

.global main                          // Note [5]
main:
    rcall   ioinit
1:  rjmp    1b                        // Note [6]

.global TIMER0_OVF_vect               // Note [7]
TIMER0_OVF_vect:
    ldi     inttmp, 256 - tmconst + fuzz
    out     IO(TCNT0), inttmp         // Note [8]

    in      intsav, IO(SREG)          // Note [9]

    sbic    IO(PORTD), SQUARE
    rjmp    1f
```

```
    sbi     IO(PORTD), SQUARE
    rjmp    2f
1:  cbi     IO(PORTD), SQUARE
2:
    out     IO(SREG), intsav
    reti

ioinit:
    sbi     IO(DDRD), SQUARE

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

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

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

    sei

    ret

.global __vector_default            // Note [10]
__vector_default:
    reti
```

Note [1] As in C programs, this includes the central processor-specific file containing the IO port definitions for the device. Note that not all include files can be included into assembler sources.

Note [2] Assignment of registers to symbolic names used locally. Another option would be to use a C preprocessor macro instead:

```
#define work 16
```

Note [3] Our bit number for the square wave output. Note that the right-hand side consists of a CPP macro which will be substituted by its value (6 in this case) before actually being passed to the assembler.

Note [4] The assembler uses integer operations in the host-defined integer size (32 bits or longer) when evaluating expressions. This is in contrast to the C compiler that uses the C type `int` by default in order to calculate constant integer expressions.

In order to get a 100 kHz output, we need to toggle the PD6 line 200000 times per second. Since we use timer 0 without any prescaling options in order to get the desired frequency and accuracy, we already run into serious timing considerations: while accepting and processing the timer overflow interrupt, the timer already continues to count. When pre-loading the `TCCNT0` register, we therefore have to account for the number of clock cycles required for interrupt acknowledge and for the instructions to reload `TCCNT0` (4 clock cycles for interrupt acknowledge, 2 cycles for the jump from the interrupt vector, 2 cycles for the 2 instructions that reload `TCCNT0`). This is what the constant `fuzz` is for.

Note [5] External functions need to be declared to be `.global`. `main` is the application entry point that will be jumped to from the ininitalization routine in `crts1200.o`.

Note [6] The main loop is just a single jump back to itself. Square wave generation itself is completely handled by the timer 0 overflow interrupt service. A `sleep` instruction (using idle mode) could be used as well, but probably would not conserve much energy anyway since the interrupt service is executed quite frequently.

Note [7] Interrupt functions can get the usual names that are also available to C programs. The linker will then put them into the appropriate interrupt vector slots. Note that they must be declared `.global` in order to be acceptable for this purpose. This will only work if `<avr/io.h>` has been included. Note that the assembler or linker have no chance to check the correct spelling of an interrupt function, so it should be double-checked. (When analyzing the resulting object file using `avr-objdump` or `avr-nm`, a name like `__vector_N` should appear, with *N* being a small integer number.)

Note [8] As explained in the section about special function registers, the actual IO port address should be obtained using the macro `_SFR_IO_ADDR`. (The AT90S1200 does not have RAM thus the memory-mapped approach to access the IO registers is not available. It would be slower than using `in` / `out` instructions anyway.)

Since the operation to reload `TCCNT0` is time-critical, it is even performed before saving `SREG`. Obviously, this requires that the instructions involved would not change any of the flag bits in `SREG`.

Note [9] Interrupt routines must not clobber the global CPU state. Thus, it is usually necessary to save at least the state of the flag bits in `SREG`. (Note that this serves as an example here only since actually, all the following instructions would not modify `SREG` either, but that's not commonly the case.)

Also, it must be made sure that registers used inside the interrupt routine do not conflict with those used outside. In the case of a RAM-less device like the AT90S1200, this can only be done by agreeing on a set of registers to be used exclusively inside the interrupt routine; there would not be any other chance to "save" a register anywhere.

If the interrupt routine is to be linked together with C modules, care must be taken to follow the register usage guidelines imposed by the C compiler. Also, any register modified inside the interrupt sevice needs to be saved, usually on the stack.

Note [10] As explained in Interrupts, a global "catch-all" interrupt handler that gets all unassigned interrupt vectors can be installed using the name `__vector_default`. This must be `.global`, and obviously, should end in a `reti` instruction. (By default, a jump to location 0 would be implied instead.)

6.4 Assembler Directives

The directives available in the assembler are described in the GNU assembler (gas) manual at `Assembler Directives`.

As gas comes from a Unix origin, its directives and overall assembler syntax is slightly different than the one being used by other assemblers. Numeric constants follow the C notation (prefix `0x` for hexadecimal constants, `0b` for binary constants), expressions use a C-like syntax.

Some common directives include:

Table 7 Assembler Directives: Sections

Section Ops	Description
`.section` *name*`,`*"flags"*`,@`*typ*	Put the following objects into named section *name*. Set section flags *flags* and section type to *typ*
`.pushsection ...` `.popsection`	Like `.section`, but also pushes the current section and subsection onto the section stack. The current section and subsection can be restored with `.popsection`.
`.subsection` *int*	Put the following code into subsection number *int* which is some integer. Subsections are located in order of increasing index within their input section. The default after switching to a new section by means of `.section` or `.pushsection` is subsection 0.

`.text`
`.data`
`.bss`
	Put the following code into the `.text` section, `.data` section or `.bss` section, respectively. The assembler knows the right section flags and section type, for example the `.text` directive is basically the same like `.section .text,"ax",@progbits`. The directives support an optional subsection argument, see `.subsection` above.

Table 8 Assembler Directives: Symbols

Symbol Ops	Description
`.global` *sym* `.globl` *sym*	Globalize symbol *sym* so that it can be referred to in other modules. When a symbol is used without prior declaration or definition, the symbol is implicitly global. The `.global` directive can also by used to refer to that symbol, so that the linker pulls in code that defines the symbol (provided such a symbol definition exists). For example, code that puts objects in the `.data` section and that assumes that the startup code initializes that area, would use `.global __do_copy_data`.

Symbol Ops	Description
`.weak` *syms*	Declare symbols *syms* as weak symbols, where *syms* is a comma-separated list of symbols. This applies only when the symbols are also defined in the same module. When the linker encounters a weak symbol that is also defined as `.global` in a different module, then the linker will use the latter without raising a diagnostic about multiple symbol definitions.
`.type` *sym,@kind*	Set the type of symbol *sym* to *kind*. Commonly used symbol types are `@function` for function symbols like `main` and `@object` for data symbols. This has an affect for disassemblers, debuggers and tools that show function / object properties.
`.size` *sym,size*	Set the size associated with symbol *sym* to expression *size*. The linker works on the level of sections, it does not even know what functions are. This directive serves book-keeping, and may be useful for debuggers, disassemblers or tools that show which function / object consumes how much memory.
`.set` *sym, expr* `.equ` *sym, expr* *sym = expr*	Set the value of symbol *sym* to the value of expression *expr*. When a global symbol is set multiple times, the value stored in the object file is the last value stored into the symbol.
`.extern`	Ignored for compatibility with other assemblers.
`.org`	Advance the location pointer to a specified offset *relative* to the beginning of the input section. The location counter cannot be moved backwards. This is a fairly pointless directive in an assembler environment that uses relocatable object files. The linker determines the final location of the objects. See the FAQ on how to relocate code to a fixed address.

Table 9 Assembler Directives: Data and Alignment

Data Ops	Description	Alias
`.byte` *list*	Allocate bytes specified by a list of comma-separated expressions.	
`.2byte` *list*	Similar to `.byte`, but for 16-bit values.	`.word`
`.4byte` *list*	Similar to `.byte`, but for 32-bit values.	`.long`
`.8byte` *list*	Similar to `.byte`, but for 64-bit values.	`.qword`
`.ascii` *"string"*	Allocate a string of characters without \0 termination.	
`.asciz` *"string"*	Allocate a \0 terminated string.	
`.float` *list*	Allocate IEEE-754 single 32-bit floating-point values specified in the comma-separated *list*.	
`.double` *list*	Same, but for IEEE-754 double 64-bit floats.	
`.space` *num[,val]*	Allocate *num* bytes with value *val* where *val* is optional and defaults to zero.	`.skip`
`.zero` *num*	Insert *num* zero bytes.	
Alignment Ops	**Description**	**Alias**
`.balign` *val*	Align the following code to *val* bytes, where *val* is an absolute expression that evaluates to a power of 2.	`.align`
`.p2align` *expo*	Align the following code to 2^{expo} bytes.	

Moreover, there is the **`.macro`** directive, which starts an assembler macro. The GNU assembler implements a powerful macro processor which even supports recursive macro definitions. For an example, see the gas documentation for `.macro`. A gas `.macro` can further be combined with C preprocessor directives. For some real-world examples, see the AVR-LibC sources `macros.inc` and `asmdef.h`.

6.5 Operand Modifiers

There are some AVR-specific operators available like `lo8()`, `hi8()`, `pm()`, `gs()` etc. For an overview see the documentation of the operand modifiers in the inline assembly Cookbook.

Example:

```
ldi  r24, lo8(gs(somefunc))
ldi  r25, hi8(gs(somefunc))
call      something
subi r24, lo8(-(my_var))
sbci r25, hi8(-(my_var))
```

This passes the address of function `somefunc` as the first parameter to function `something`, and *adds* the address of variable `my_var` to the 16-bit return value of `something`.

7 Inline Assembler Cookbook

AVR-GCC
Inline Assembler Cookbook

7.1 About this Document

The GNU C/C++ compiler for AVR RISC processors offers to embed assembly language code into C/C++ programs. This cool feature may be used for manually optimizing time critical parts of the software, or to use specific processor instructions which are not available in the C language.

It's assumed that you are familiar with writing AVR assembler programs, because this is not an AVR assembler programming tutorial. It's not a C/C++ tutorial either.

Note that this document does not cover files written completely in assembly language, refer to AVR-LibC and Assembler Programs for this.

This document describes version 4.7 of the compiler or newer.

Herne, 17th of May 2002 Harald Kipp harald.kipp-at-egnite.de

7.2 The Anatomy of a GCC asm Statement

A GCC inline assembly statement starts with the keyword `asm`, `__asm` or `__asm__`, where the first one is not available in strict ANSI mode.

In its simplest form, the inline assembly statement has no operands and injects just one instruction into the code stream, like in

```
__asm ("nop");
```

In its generic form, an asm statements can have one of the following three forms:

A simple asm without operands `__asm (code-string);`

`code-string` is a string literal that will be added as is into the generated assembly code. This even applies to the % character. The only replacement is that \n and \t are interpreted as newline resp. TAB character.

This type of asm statement may occur at top level, outside any function as global asm. When its placement relative to functions is important, consider `-fno-toplevel-reorder`.

An asm with operands `__asm volatile (code-string : output-operands : input-operands : clobbers);`

This is the most widely used form of an asm statement. It must be located in a function.

`output-operands`, `input-operands` and `clobbers` are comma-separated lists of operands resp. clobber specifications. Any of them may be empty, for example when the asm has no outputs. At least one : (colon) must be present, otherwise it will be a simple asm without operands and without `%` replacements.

An asm goto statement `__asm goto (code-string : : input-operands : clobbers : labels);`

Like the asm above, but `labels` is a comma-separated list of C/C++ code labels which would be valid in a `goto` statement. And `output-operands` must be empty, because it is impossible to generate output reloads after the code has transferred control to one of the labels.
As there are no output operands, asm goto is implicitly volatile. When `volatile` is specified explicitly, the `goto` keyword may be placed after or before the `volatile`.

Notes on the various parts:

Volatility Keyword `volatile` is optional and means that the asm statement has side effects that are not expressed in terms of the operands or clobbers. The asm statement must not be optimized away or reordered with respect to other volatile statements like volatile memory accesses or other volatile asm.

Any asm statement without `output-operands` is implicitly volatile.

A non-volatile asm statement with output operands that are all unused may be optimized away when all output operands are unused.

Instead of `volatile`, `__volatile` or `__volatile__` can be used.

`code-string` A string literal that contains the code that is to be injected in the assembly code generated by the compiler. `%-expressions` are replaced by the string representations of the operands, and the number of lines is determined to estimate the code size of the asm.
Apart from that, **the compiler does not analyze the code provided in the code template.**
This means that the code appears to the compiler *as if it was executed in one parallel chunk, all at once*. It is important to keep that in mind, in particular for cases where input and output operands may overlap.

`output-operands`

`input-operands` A comma-separated list of operands, which may take the following forms. In any case, the first operand can be referred to as `"%0"` in `code-string`, the second one as `"%1"` etc.

`"constraints" (expr)` `expr` is a C expression that's an input or output (or both) to the asm statement. An output expression must be an lvalue, i.e. it must be valid to assign a value to it.
`"constraints"` is a string literal with constraints and constraint modifiers. For example, constraint `"r"` stands for *general-purpose register*. A simple input operand would be

```
"r" (value + 1)
```

The compiler computes `value + 1` and supplies it in some general-purpose register R2...R31. In many cases, an upper d-register R16...R31 is required for instructions like `LDI` or `ANDI`. A respective output operand specification is

```
"=d" (result)
```

Notice that this operand may overlap with input operands!
When an operand is written before all input operands are consumed, then in almost all cases the output operand requires an early-clobber modifier `&` so that it won't overlap with any input operand:

```
"=&d" (result)
```

An operand that's both an output and an input can be expressed with the `+` constraint modifier:

```
"+d" (result)
```

Such an operand is both output and input, and hence it won't overlap with other operands.

`[name] "constraints" (expr)` Like above. In addition, a named operand can be referred to as `%[name]` in `code-string`. This is useful in long asm statements with many operands.

`clobbers` A comma-separated list of string literals like `"16"`, "r16" or "memory".

The first two clobbers mean that the asm destroys register R16. Only the lower-case form is allowed, and register names like Z are not recognized.

`"memory"` means that the asm touches memory in some way. When the asm writes to some RAM location for example, the compiler must not optimize RAM accesses across the asm because the memory may change.

Clobbering `__tmp_reg__` by means of `"r0"` has no effect, but such a clobber may be added to indicate to the reader that the asm clobbers R0.

Clobbering `__zero_reg__` by means of `"r1"` has no effect. When the asm destroys the zero register, for example by means of a `MUL` instruction, then the code must restore the register at the end by means of `"clr __zero_reg__"`

The size of an asm The code size of an asm statement is the number of lines multiplied by 4 bytes, the maximal possible AVR instruction length. The length is needed when (conditional) jumps cross the asm statement in order to compute (upper bounds for) jump offsets of PC-relative jumps.

The number of lines is one plus the number of line breaks in `code-string`. These may be physical line breaks from `\n` characters and logical line breaks from `$` characters.

Before we start with the first examples, we list all the bells and whistles that can be used to compose an inline assembly statement: special sequences, constraints, constraint modifiers, print modifiers and operand modifiers.

7.3 Special Sequences

There are special sequences that can be used in the assembly template.

Table 10 Inline asm Special Sequences

Sequence	Description
`__SREG__`	The I/O address of the status register SREG at 0x3F
`__tmp_reg__`	The temporary register R0 (R16 on reduced Tiny)
`__zero_reg__`	The zero register R1, always zero (R17 on reduced Tiny)
`$`	A logical line separator, used to separate multiple instruction in one physical line
`\n`	A physical newline, used to separate multiple instructions
`\t`	A TAB character, can be used for better legibility of the generated asm
`\"`	A " character (double quote)
`\\`	A \ character (backslash)
`%%`	A % charater (percent)
`%~`	`"r"` or `""`, used to construct `call` or `rcall` by means of `"%~call"`, depending on the architecture
`%!`	`""` or `"e"`, used to construct indirect calls like `icall` or `eicall` by means of `"%!icall"`, depending on the architecture
`%=`	A number that's unique for the compilation unit and the respective inline asm code, used to construct unique labels
Comment	**Description**
`; text`	A single-line assembly comment that extends to the end of the physical line
`/* text */`	A multi-line C comment

- Moreover, the following I/O addresses are defined provided the device supports the respective SFR: `__SP_L__`, `__SP_H__`, `__CCP__`, `__RAMPX__`, `__RAMPY__`, `__RAMPZ__`, `__RAMPD__`.
- Register `__tmp_reg__` may be freely used by inline assembly code and need not be restored at the end of the code.
- Register `__zero_reg__` contains a value of zero. When that value is destroyed, for example by a `MUL` instruction, its value has to be restored at the end of the code by means of

```
clr __zero_reg__
```

- In inline asm without operands (i.e without a single colon), a `%` will always insert a single `%`. No `%-codes` are available.

Sequences like `__SREG__` are not evaluated as part of the inline asm, they are just copied to the asm code as they are. At the top of each assembly file, the compiler prints definitions like

```
__SREG__ = 0x3f
```

so that they can also be used in inline assembly.

7.4 Constraints

The most up-to-date and detailed information on constraints for the AVR can be found in the `avr-gcc Wiki`.

Table 11 Inline asm Operand Constraints

Constraint	Registers	Range
a	Simple upper registers that support FMUL	R16 ... R23
b	Base pointer registers that support LDD, STD	Y, Z (R28 ... R31)
d	Upper registers	R16 ... R31
e	Pointer registers that support LD, ST	X, Y, Z (R26 ... R31)
l	Lower registers	R2 ... R15
r	Any register	R2 ... R31
w	Upper registers that support ADIW	R24 ... R31
x	X pointer registers	R26, R27
y	Y pointer registers	R28, R29
z	Z pointer registers	R30, R31
Constraint	**Constant**	**Range**
I	6-bit unsigned integer constant	0 to 63
J	6-bit negative integer constant	-63 to 0
M	8-bit unsigned integer constant	0 to 255
n	Integer constant	
i	Immediate value known at link-time, like the address of a variable in static storage	
EF	Floating-point constant	
Ynn	Fixed-point or integer constant	
Constraint	**Explanation**	**Notes**
m	A memory location	
X	Any valid operand	
0 ... 9	Matches the respective operand number	

- Constraints without a modifier specify input operands.
- Constraints with a modifier specify output operands.
- More than one constraint like in `"rn"` specifies the union of the specified constraints; `"r"` and `"n"` in this case.
- All constraints listed above are single-letter constraints, except `Ynn` which is a 3-letter constraint.

Constraint modifiers are:

Table 12 Constraint Modifiers

Modifier	Meaning
=	Output-only operand. Without & it may overlap with input operands
+	Output operand that's also an input
=&	"Early-clobber". Register should be used for output only and won't overlap with any input operand(s)

The selection of the proper constraint depends on the range of the constants or registers, which must be acceptable to the AVR instruction they are used with. The C compiler doesn't check any line of your assembler code. But it is able to check the constraint against your C expression. However, if you specify the wrong constraints, then the compiler may silently pass wrong code to the assembler. And, of course, the assembler will fail with some cryptic output or internal errors, or in the worst case wrong code may be the result.

For example, if you specify the constraint `"r"` and you are using this register with an `ORI` instruction, then the compiler may select any register. This will fail if the compiler chooses `R2` to `R15`. (It will never choose `R0` or `R1`, because these are uses for special purposes.) That's why the correct constraint in that case is "d". On the other hand, if you use the constraint "M", the compiler will make sure that you don't pass anything else but an 8-bit unsigned integer value known at compile-time.

The following table shows all AVR assembler mnemonics which require operands, and the related constraints.

Table 13 AVR Instructions and Constraints

Mnemonic	Constraints		Mnemonic	Constraints
adc	r,r		add	r,r
adiw	w,I		and	r,r
andi	d,M		asr	r
bclr	I		bld	r,I
brbc	I,label		brbs	I,label
bset	I		bst	r,I
call	i		cbi	I,I
cbr	d,I		clr	r
com	r		cp	r,r
cpc	r,r		cpi	d,M
cpse	r,r		dec	r
elpm	r,z		eor	r,r
fmul	a,a		fmuls	a,a
fmulsu	a,a		in	r,I
inc	r		jmp	i
lac	z,r		las	z,r
lat	z,r		ld	r,e
ldd	r,b		ldi	d,M
lds	r,i		lpm	r,z
lsl	r		lsr	r
mov	r,r		movw	r,r
mul	r,r		muls	r,r
mulsu	a,a		neg	r
or	r,r		ori	d,M
out	I,r		pop	r
push	r		rcall	i
rjmp	i		rol	r
ror	r		sbc	r,r
sbci	d,M		sbi	I,I
sbic	I,I		sbiw	w,I
sbr	d,M		sbrc	r,I
sbrs	r,I		ser	d
st	e,r		std	b,r
sts	i,r		sub	r,r
subi	d,M		swap	r
tst	r		xch	z,r

7.5 Print Modifiers

The %-operands in the inline assembly template can be adjusted by special print-modify characters. The one-letter modifier follows the % and precedes the operand number like in "%a0", or precedes the name in named operands like in "%a[address]".

Table 14 Inline asm Print Modifiers

Modifier	Number of Arguments	Explanation	Suitable Constraints
%a0	1	Print pointer register as address X, Y or Z, like in "LD r0, %a0+"	x, y, z, b, e
%i0	1	Print compile-time RAM address as I/O address, like in "OUT %i0, r0" with argument "n"(&SREG)	n
%n0	1	Print the negative of a compile-time integer constant	n
%r0	1	Print the register number of a register, like in "CLR %r0+7" for the MSB of a 64-bit register	reg
%x0	1	Print a function name without gs() modifier, like in "%~CALL %x0" with argument "s"(main)	s
%A0	1	Add 0 to the register number (no effect)	reg
%B0	1	Add 1 to the register number	reg
%C0	1	Add 2 to the register number	reg
%D0	1	Add 3 to the register number	reg
%T0%t1	2	Print the register that holds bit number %1 of register %0	reg + n
%T0%T1	2	Print operands suitable for BLD/BST, like in "BST %T0%T1", including the required ,	reg + n

- Register constraints are: r, d, w, x, y, z, b, e, a, l.

7.6 Operand Modifiers

Table 15 Assembly Code Operand Modifiers

Modifier	Explanation	Purpose
lo8()	1st Byte of a link-time constant, bits 0...7	Getting partsof a byte-address
hi8()	2nd Byte of a link-time constant, bits 8...15	
hlo8()	3rd Byte of a link-time constant, bits 16...23	
hhi8()	4th Byte of a link-time constant, bits 24...31	
hh8()	Same like hlo8	
pm_lo8()	1st Byte of a link-time constant divided by 2, bits 1...8	Getting partsof a word-address
pm_hi8()	2nd Byte of a link-time constant divided by 2, bits 9...16	
pm_hh8()	3rd Byte of a link-time constant divided by 2, bits 17...24	
pm()	Link-time constant divided by 2 in order to get a **p**rogram **m**emory (word) addresses, like in lo8(pm(main))	Word-address
gs()	Function address divided by 2 in order to get a (word) addresses, like in lo8(gs(main)). **G**enerate **s**tub (trampoline) as needed. This is required to calculate the address of a code label on devices with more than 128 KiB of program memory that's supposed to be used in EICALL. For rationale, see the GCC documentation. On devices with less program memory, gs() behaves like pm()	Function address for [E]ICALL

When the argument of a modifier is not computable at assembler-time, then the assembler has to encode the expression in an abstract form using RELOCs. Consequence is that only a very limited number of argument expressions is supported when they are not computable at assembler-time.

7.7 Examples

Some examples show the assembly code as generated by the compiler. It's the code from the `.s` files as generated with option `-save-temps`. Adding the high-level source to the generated assembly can be turned on with `-fverbose-asm` since GCC v8.

7.7.1 Swapping Nibbles

The fist example uses the `swap` instruction to swap the nibbles of a byte. Input and output of `swap` are located in the same general purpose register. This means the input operand, operand 1 below, must be located in the same register(s) like operand 0, so that the right constraint for operand 1 is `"0"`:

```
asm ("swap" : "=r" (value) : "0" (value));
```

All side effects of the code are described by the constraints and the clobbers, so that there is no need for this asm to be volatile. In particular, this asm may be optimized out when the output value is unused.
A shorter pattern to state that `value` is both input and output is by means of constraint modifier `+`

```
asm ("swap" : "+r" (value));
```

7.7.2 Swapping Bytes

Swapping nibbles was a piece of cake, so let's swap the bytes of a 16-bit value. In order to access the constituent bytes of the 16-bit input and output values, we use the print modifiers `%A` and `%B`.

The asm is placed in a small C test case so that we can inspect the resulting assembly code as generated by the compiler with `-save-temps`.

```
void callee (int, int);

void func (int param)
{
    int swapped;

    asm ("mov %A0, %B1" "\n\t"
         "mov %B0, %A1"
         : "=r" (swapped) : "r" (param));

    callee (param, swapped);
}
```

The `"\n\t"` sequence adds a line feed that is required between the two instructions, and a TAB to align the two instructions in the generated assembly. There is no `"\n\t"` after the last instruction because that would just increase the size of the asm.
The generated assembly works as expected. The compiler wraps it in `#APP` / `#NOAPP` annotations:

```
func:
/* #APP */
    mov r22, r25     ;  swapped, param
    mov r23, r24     ;  swapped, param
/* #NOAPP */
    jmp callee
```

Wrong! **While the generated code above is correct, the inline asm itself is not!**
We see this with a slightly adjusted test case where the arguments of `callee` have been swapped, but that uses the same inline asm:

```
void func (int param)
{
    int swapped;

    asm ("mov %A0, %B1" "\n\t"
         "mov %B0, %A1"
         : "=r" (swapped) : "r" (param));

    callee (swapped, param);
}
```

The result is the following assembly:

```
func:
```

```
    movw r22,r24
/* #APP */
    mov r24, r25     ;  swapped, param
    mov r25, r24     ;  swapped, param
/* #NOAPP */
    jmp callee
```

which is obviously wrong, because after the code from the inline asm, the low byte of `swapped` and the high byte will always have the same value of `r25`.

The reason is that the output operand overlaps the input, *and* the output is changed before all of the input operands are consumed. This is a so-called *early-clobber* situation. There are two possible solutions to this predicament:

- Mark the output operand with the early-clobber constraint modifier:

```
asm ("mov %A0, %B1" "\n\t"
     "mov %B0, %A1"
     : "=&r" (swapped) : "r" (param));
```

- Use constraints and a code sequence that expect input and output in the same registers:

```
asm ("eor %A0, %B0" "\n\t"
     "eor %B0, %A0" "\n\t"
     "eor %A0, %B0"
     : "=r" (swapped) : "0" (param));
```

7.7.3 Accessing Memory

Accessing memory requires that the AVR instructions that perform the memory access are provided with the appropriate memory address.

1. The address can be provided directly, like __SREG__, 0x3f, as a symbol, or as a symbol plus a constant offset.
2. Provide the address by means of an inline asm operand.

Approach 1 is simpler as it does not require an asm operand, while approach 2 is in many cases more powerful because macros defined per, say, `#include <avr/io.h>` can be used as operands, whereas such headers are not included in the assembly code as generated by the compiler.

Reading a SFR like `PORTB` can be performed by

```
asm volatile ("in %0, %1" : "=r" (result) : "I" _SFR_IO_ADDR (PORTB));
```

Macro `_SFR_IO_ADDR` is provided by `avr/sfr_defs.h` which is included by `avr/io.h`.

Since GCC v4.7, print modifier `%i` is supported, which prints RAM addresses like `& PORTB` as an I/O address:

```
asm volatile ("in %0, %i1" : "=r" (result) : "I" (& PORTB));
```

When the address is not an I/O address, then `LDS` or `LD` must be used, depending on whether the address is known at link-time or only at run-time. For example, the following macro provides the functionality to clear an SFR. The code discriminates between the possibilities that

- The SFR address is known at compile-time and is an I/O address.
- The SFR address is known at compile-time but is not in the I/O range.
- The SFR address is not known at compile-time.

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

#define CLEAR_REG(sfr)                            \
do {                                              \
  if (__builtin_constant_p (& (sfr))              \
      && _SFR_IO_REG_P (sfr))                     \
    asm volatile ("out %i0, __zero_reg__"         \
                  :: "I" (& (sfr)) : "memory"); \
  else if (__builtin_constant_p (& (sfr)))        \
    asm volatile ("sts %0, __zero_reg__"          \
                  :: "n" (& (sfr)) : "memory"); \
  else                                            \
    asm volatile ("st %a0, __zero_reg__"          \
                  :: "e" (& (sfr)) : "memory"); \
} while (0)
```

The last case with constraint `"e"` works because `&sfr` is a 16-bit value, and 16-bit values (and larger) start in even registers. Therefore, the address will be located in R27:R26, R29:R28 or in R31:R30, which print modifier `%a` will print as X, Y or Z, respectively. The address will never end up in, say, R30:R29.

The test case

```
void clear_3_regs (uint8_t volatile *psfr)
{
    CLEAR_REG (PORTB);
    CLEAR_REG (UDR0);
    CLEAR_REG (*psfr);
}
```

compiles for ATmega328 and with optimization turned on to

```
clear_3_regs:
    movw r30,r24
/* #APP */
    out 0x5, __zero_reg__
    sts 198, __zero_reg__
    st Z,    __zero_reg__   ;  psfr
/* #NOAPP */
    ret
```

As `__builtin_constant_p` is used to infer whether the address of the SFR is known at compile-time, extra care must be taken when the functionality is implemented as an inline function:

```
static inline __attribute__((__always_inline__))
void clear_reg (uint8_t volatile *psfr)
{
  // !!! The following cast is required to make __builtin_constant_p
  // !!! work as expected in the inline function.
  uintptr_t addr = (uintptr_t) psfr;

  if (__builtin_constant_p (addr)
      && _SFR_IO_REG_P (* psfr))
    asm volatile ("out %i0, __zero_reg__"
                  :: "I" (addr) : "memory");
  else if (__builtin_constant_p (addr))
    asm volatile ("sts %0, __zero_reg__"
                  :: "n" (addr) : "memory");
  else
    asm volatile ("st %a0, __zero_reg__"
                  :: "e" (addr) : "memory");
}

void clear_3_pregs (uint8_t volatile *psfr)
{
  clear_reg (& PORTB);
  clear_reg (& UDR0);
  clear_reg (psfr);
}
```

Casting the address `psfr` to an integer type in the inline function is required so that the compiler will recognize constant addresses.
Also notice that we have to pass the *address of the SFR* to the inline function. Passing the SFR directly like in the marco approach won't work for obvious reasons.

7.7.4 Accessing Bytes of wider Expressions

Finally, an example that atomically increments a 16-bit integer. The code is wrapped in `IN SREG` / `CLI` / `OUT SREG` to make it atomic. It reads the 16-bit value `data` from its absolute address, increments it and then writes it back:

```
uint16_t volatile data;

void inc_data (void)
{
    uint16_t tmp;
    asm volatile ("in __tmp_reg__, __SREG__"   "\n\t"
                  "cli"                         "\n\t"
                  "lds %A[temp], %[addr]"       "\n\t"
                  "lds %B[temp], %[addr]+1"     "\n\t"
#ifdef __AVR_TINY__
                  // Reduced Tiny does not have ADIW.
                  "subi %A[temp], lo8(-1)"      "\n\t"
                  "sbci %B[temp], hi8(-1)"      "\n\t"
#else
                  "adiw %[temp], 1"             "\n\t"
#endif
                  "sts %[addr]+1, %B[temp]"     "\n\t"
                  "sts %[addr],   %A[temp]"     "\n\t"
                  "out __SREG__, __tmp_reg__"
#ifdef __AVR_TINY__
                  // No need to restrict tmp to a "w" register. And on
                  // avr-gcc v13.2 and older, "w" contains no regs.
                  : [temp] "=d" (tmp), "+m" (data)
#else
                  : [temp] "=w" (tmp), "+m" (data)
#endif
                  : [addr] "i" (& data));
}
```

Notice there are three different ways required to access the different bytes of the involved 16-bit entities:

- For the 16-bit general purpose register %[temp], print modifiers `%A` and `%B` are used.
- For the 16-bit value `data` in static storage, `%[addr]+1` is used to access the high byte. The resulting expression `data+1` is computable at link-time and evaluated by the linker.
- In the compilation variant for Reduced Tiny, the bytes of the 16-bit subtrahend `-1` are accessed with the operand modifiers `lo8` and `hi8` that are evaluated by the assembler because `-1` is known at assembler-time.

`data` is located in static storage, hence its address is known to the linker and fits constraint `"i"`.

The sole purpose of operand `"+m" (data)` is to describe the effect of the asm on data memory: It changes `data`. Notice that there is no `"memory"` clobber, because that operand already describes all memory side effects, and it does this in a less intrusive way than a catch-all `"memory"`. The operand is not used in the asm template; but in principle it would be possible to use it as operand with `LDS` and `STS` instead of operand `[addr] "i" (& data)`. However, there are many situations where a memory operand constrained by `"m"` takes a form that cannot be used with AVR instructions because there are no matching print modifiers, or because it is not known a priori what specific form the memory operand takes. In such cases, one would take the address of the operand and supply it as address in a pointer register to the inline asm. The compiler generates the required instructions for address computation, and the inline asm knows that it can use `LD` and `ST`.

7.7.5 Jumping and Branching

When an inline asm contains jumps, then it also requires labels. When the label is inside the asm, then care must be taken that the label is unique in the compilation unit even when the inline asm is used multiple times, e.g. when the code is located in an unrolled loop or a function has multiple incarnations due to cloning, or simply because a macro or inline function that contains an asm statement is used more than once.
There are two kinds of labels that can be used:

- Local labels of the form `n:` where *n* is some (small, non-negative) number. They can be targeted by means of `nb` or `nf`, depending on whether the jump direction is **b**ackwards or **f**orwards. Such a numeric labels may be present more than once. The taken label is the first one with the specified number in the respective direction:

```
// Loop until bit PORTB.7 is set.
asm volatile ("1: sbrs %i[sfr], %[bitno]"  "\n\t"
              "rjmp 1b"
              :: [sfr] "I" (& PORTB), [bitno] "n" (PB7));
```

- Local labels that contain the sequence %= which yields some number that's unique amongst all asm incarnations in the respective compilation unit:
```
// Loop until bit PORTB.7 is set.
asm volatile (".Loop.%=: sbrs %i[sfr], %[bitno]"  "\n\t"
              "rjmp .Loop.%="
              :: [sfr] "I" (& PORTB), [bitno] "n" (PB7));
```

Which form is used is a matter of taste. In practice, the first variant is often preferred in short sequences, whereas the second form is usually seen in longer algorithms.

For labels that are defined in the surrounding C/C++ code, asm goto has to be used. The print modifier `%x0` prints `panic` as a raw label, not as `gs(panic)` like it would be the case with `%0`.
```
int main (void)
{
    asm goto ("tst __zero_reg__" "\n\t"
              "brne %x0"
              :::: panic);
    /* ...Application code here... */
    return 0;
panic:
    // __zero_reg__ is supposed to contain 0, but doesn't.
    return 1;
}
```

This assumes that the jump offset can be encoded in the `brne` instruction in all situations. When static analysis cannot prove that the jump offset fits, then a jumpity jump has to be used:
```
asm goto ("tst   __zero_reg__" "\n\t"
          "breq  1f"           "\n\t"
          "%~jmp %x0"          "\n"
          "1: ;; all fine"
          :::: panic);
```

Sequence `"%~jmp"` yields `"rjmp"` or `"jmp"` depending on the architecture. Notice that a `jmp` can be relaxed to an `rjmp` with option `-mrelax` provided the jump offset fits.

7.8 Binding local Variables to Registers

One use of GCC's `asm` keyword is to bind local register variables to hardware registers.
Such bindings of local variables to registers are only guaranteed during inline asm which has these variables as operands.

7.8.1 Interfacing non-ABI Functions

Suppose we want to interface a non- ABI assembly function `mul_8_16` that multiplies R24 with R27:R26, clobbers R0, R1 and R25, and returns the 24-bit result in R20:R19:R18. One way to implement such an interface would be to provide an assembly function that performs the required copying and call to `mul_8_16`. Such a function would destroy some of the performance gain obtained by using assembly for `mul_8_16`: Additional copying back and forth and extra `CALL` and `RET` instructions.

The compiler comes to the rescue. We can bind local variables to the required registers:
```
extern void mul_8_16 (void); // Non-ABI function. Don't call in C/C++!

static inline __attribute__((__always_inline__))
__uint24 mul_8_16_gccabi (uint8_t val8, uint16_t val16)
{
    register uint8_t r24 __asm("r24") = val8;
    register __uint24 r18 __asm("r18");

    asm ("%~call %x[func]"  "\n\t"
         "clr    __zero_reg__"
         : "=r" (r18)
         : "r" (r24), "x" (val16), [func] "i" (mul_8_16)
         : "r25", "r0");

    return r18;
}
```

- The 8-bit parameter is bound to R24, and the 24-bit return value is bound to R18...R20.
- The `register` keyword is mandatory.
- The hard register is specified as a string literal for the lower case register name or register number, like `"18"` or `"r18"`. Specifications like `"R18"`, `18` or `"Z"` are not supported.
- The 16-bit parameter of `mul_8_16` happens to be required in R27:R26, which is the X register for which there is register constraint `"x"`. Therefore, no register binding is required for `val16`.
- As `mul_8_16` clobbers the zero register R1, it has to be restored by means of
```
clr __zero_reg__
```
- The asm is pure arithmetic and hence not volatile. (It might be advisable to make it volatile anyway, so that it won't be reorderd across `sei()` or `cli()` instructions.)

Let's have a look at how this performs in a test case:
```
void use_mul_8_16_gccabi (uint8_t val, uint8_t a, uint8_t b)
{
    if (mul_8_16_gccabi (val, a * b) >= 0x2010)
        __builtin_abort();
}
```

For ATmega8 we get the following assembly:
```
use_mul_8_16_gccabi:
    mul  r22,r20
    movw r26,r0
    clr  __zero_reg__
/* #APP */
    rcall mul_8_16
    clr   __zero_reg__
/* #NOAPP */
    cpi  r18,16
    sbci r19,32
    cpc  r20,__zero_reg__
    brlo .L1
    rcall abort
.L1:
    ret
```

No superfluous register moves. Great!

7.9 Specifying the Assembly Name of Static Objects

Sometimes, it is desirable to use a different name for an object or function rather than the (mangled) name from the C/C++ implementation. Just add an asm specifier with the desired name as a string literal at the end of the declaration.

For example, this is how `avr/eeprom.h` implements the `eeprom_read_double()` function:
```
#if __SIZEOF_DOUBLE__ == 4
double eeprom_read_double (const double*) __asm("eeprom_read_dword");
#elif __SIZEOF_DOUBLE__ == 8
double eeprom_read_double (const double*) __asm("eeprom_read_qword");
#endif
```

- It uses the implementation of `eeprom_read_dword` for `eeprom_read_double`, provided `double` is a 32-bit type.
- It uses the implementation of `eeprom_read_qword` for 64-bit doubles.

7.10 What won't work

GCC inline asm has some limitations.

7.10.1 Setting a Register on one asm and using it in a different one

Sequences like the following are not supposed to work:

```
char var;

void set_var (char c)
{
    __asm ("inc r24");
    __asm ("sts var, r24");
}
```

- There is no guarantee whatsoever that the value in R24 will survive from one asm to the next. Such code might work in many situations, but it is still wrong and the compiler may very well put instructions bewtween the asm statements that change R24 prior to the first asm and also between the asm statements.
- R24 is changed without noticing the compiler. When R24 contains other data, then that data will be trashed.

A correct code would be

```
__asm ("inc %0"     "\n\t"
       "sts var, %0"
       :: "r" (c) : "memory");
```

or

```
__asm ("inc %1"     "\n\t"
       "sts %0, %1"
       : "=m" (var) : "r" (c));
```

7.10.2 Letting an Operand cross the Boundaries of the Y Register

It is not possible to bind a value to a local register variable that crosses the boundaries of the Y register. For example, trying to bind a 32-bit value to R31:R28 by means of

```
register uint32_t r28 __asm ("28");
```

will result in an error message like

```
error: register specified for ’r28’ isn’t suitable for data type
```

Similarly, an operand described by a constraint will be located either completely below the Y register, as part of Y register, or above it.

7.10.3 Using Matching Constraints "=0"..."=9" with Output Operands

Suppose we want an inline asm that returns the low byte of a 16-bit value `val16`:

```
asm ("" : "=1" (lo8) : "r" (val16));
```

The diagnostic will be:

```
error: matching constraint not valid in output operand
```

8 How to Build a Library

8.1 Introduction

So you keep reusing the same functions that you created over and over? Tired of cut and paste going from one project to the next? Would you like to reduce your maintenance overhead? Then you're ready to create your own library! Code reuse is a very laudable goal. With some upfront investment, you can save time and energy on future projects by having ready-to-go libraries. This chapter describes some background information, design considerations, and practical knowledge that you will need to create and use your own libraries.

8.2 How the Linker Works

The compiler compiles a single high-level language file (C language, for example) into a single object module file. The linker (ld) can only work with object modules to link them together. Object modules are the smallest unit that the linker works with.

Typically, on the linker command line, you will specify a set of object modules (that has been previously compiled) and then a list of libraries, including the Standard C Library. The linker takes the set of object modules that you specify on the command line and links them together. Afterwards there will probably be a set of "undefined references". A reference is essentially a function call. An undefined reference is a function call, with no defined function to match the call.

The linker will then go through the libraries, in order, to match the undefined references with function definitions that are found in the libraries. If it finds the function that matches the call, the linker will then link in the object module in which the function is located. This part is important: the linker links in THE ENTIRE OBJECT MODULE in which the function is located. Remember, the linker knows nothing about the functions internal to an object module, other than symbol names (such as function names). The smallest unit the linker works with is object modules.

When there are no more undefined references, the linker has linked everything and is done and outputs the final application.

8.3 How to Design a Library

How the linker behaves is very important in designing a library. Ideally, you want to design a library where only the functions that are called are the only functions to be linked into the final application. This helps keep the code size to a minimum. In order to do this, with the way the linker works, is to only write one function per code module. This will compile to one function per object module. This is usually a very different way of doing things than writing an application!

There are always exceptions to the rule. There are generally two cases where you would want to have more than one function per object module.

The first is when you have very complementary functions that it doesn't make much sense to split them up. For example, malloc() and free(). If someone is going to use malloc(), they will very likely be using free() (or at least should be using free()). In this case, it makes more sense to aggregate those two functions in the same object module.

The second case is when you want to have an Interrupt Service Routine (ISR) in your library that you want to link in. The problem in this case is that the linker looks for unresolved references and tries to resolve them with code in libraries. A reference is the same as a function call. But with ISRs, there is no function call to initiate the ISR. The ISR is placed in the Interrupt Vector Table (IVT), hence no call, no reference, and no linking in of the ISR. In order to do this, you have to trick the linker in a way. Aggregate the ISR, with another function in the same object module, but have the other function be something that is required for the user to call in order to use the ISR, like perhaps an initialization function for the subsystem, or perhaps a function that enables the ISR in the first place.

8.4 Creating a Library

The librarian program is called `ar` (for "archiver") and is found in the GNU Binutils project. This program will have been built for the AVR target and will therefore be named `avr-ar`.

The job of the librarian program is simple: aggregate a list of object modules into a single library (archive) and create an index for the linker to use. The name that you create for the library filename must follow a specific pattern: lib*name*.a. The *name* part is the unique part of the filename that you create. It makes it easier if the *name* part relates to what the library is about. This *name* part must be prefixed by "lib", and it must have a file extension of .a, for "archive". The reason for the special form of the filename is for how the library gets used by the toolchain, as we will see later on.

Note

The filename is case-sensitive. Use a lowercase "lib" prefix, and a lowercase ".a" as the file extension.

The command line is fairly simple:

```
avr-ar rcs <library name> <list of object modules>
```

The `r` command switch tells the program to insert the object modules into the archive with replacement. The `c` command line switch tells the program to create the archive. And the `s` command line switch tells the program to write an object-file index into the archive, or update an existing one. This last switch is very important as it helps the linker to find what it needs to do its job.

Note

The command line switches are case sensitive! There are uppercase switches that have completely different actions.

MFile and the WinAVR distribution contain a Makefile Template that includes the necessary command lines to build a library. You will have to manually modify the template to switch it over to build a library instead of an application.

See the GNU Binutils manual for more information on the `ar` program.

8.5 Using a Library

To use a library, use the `-l` switch on your linker command line. The string immediately following the `-l` is the unique part of the library filename that the linker will link in. For example, if you use:

```
-lm
```

this will expand to the library filename:

```
libm.a
```

which happens to be the math library included in AVR-LibC.

If you use this on your linker command line:

```
-lprintf_flt
```

then the linker will look for a library called:

```
libprintf_flt.a
```

This is why naming your library is so important when you create it!

The linker will search libraries in the order that they appear on the command line. Whichever function is found first that matches the undefined reference, it will be linked in.

There are also command line switches that tell GCC which directory to look in (`-L`) for the libraries that are specified to be linke in with `-l`.

See the GNU Binutils manual for more information on the GNU linker (ld) program.

9 Benchmarks

The results below can only give a rough estimate of the resources necessary for using certain library functions. There is a number of factors which can both increase or reduce the effort required:

- Expenses for preparation of operands and their stack are not considered.
- In the table, the size includes all additional functions (for example, function to multiply two integers) but they are only linked from the library.
- Expenses of time of performance of some functions essentially depend on parameters of a call, for example, qsort() is recursive, and sprintf() receives parameters in a stack.
- Different versions of the compiler can give a significant difference in code size and execution time. For example, the dtostre() function, compiled with avr-gcc 3.4.6, requires 930 bytes. After transition to avr-gcc 4.2.3, the size become 1088 bytes.

9.1 A few of libc functions.

Avr-gcc version is 4.7.1

The size of function is given in view of all picked up functions. By default AVR-LibC is compiled with `-mcall-prologues` option. In brackets the size without taking into account modules of a prologue and an epilogue is resulted. Both of the size can coincide, if function does not cause a prologue/epilogue.

Function	Units	Avr2	Avr25	Avr4
atoi ("12345")	Flash bytes Stack bytes MCU clocks	82 (82) 2 155	78 (78) 2 149	74 (74) 2 149
atol ("12345")	Flash bytes Stack bytes MCU clocks	122 (122) 2 221	118 (118) 2 219	118 (118) 2 219
dtostre (1.2345, s, 6, 0)	Flash bytes Stack bytes MCU clocks	1116 (1004) 17 1247	1048 (938) 17 1105	1048 (938) 17 1105
dtostrf (1.2345, 15, 6, s)	Flash bytes Stack bytes MCU clocks	1616 (1616) 38 1634	1508 (1508) 38 1462	1508 (1508) 38 1462
itoa (12345, s, 10)	Flash bytes Stack bytes MCU clocks	110 (110) 2 879	102 (102) 2 875	102 (102) 2 875
ltoa (12345L, s, 10)	Flash bytes Stack bytes MCU clocks	134 (134) 2 1597	126 (126) 2 1593	126 (126) 2 1593
malloc (1)	Flash bytes Stack bytes MCU clocks	768 (712) 6 215	714 (660) 6 201	714 (660) 6 201
realloc ((void *)0, 1)	Flash bytes Stack bytes MCU clocks	1284 (1172) 18 305	1174 (1064) 18 286	1174 (1064) 18 286
qsort (s, sizeof(s), 1, cmp)	Flash bytes Stack bytes MCU clocks	1252 (1140) 42 21996	1022 (912) 42 19905	1028 (918) 42 17541
sprintf_min (s, "%d", 12345)	Flash bytes Stack bytes MCU clocks	1224 (1112) 53 1841	1092 (982) 53 1694	1088 (978) 53 1689

sprintf (s, "%d", 12345)	Flash bytes Stack bytes MCU clocks	1614 (1502) 58 1647	1476 (1366) 58 1552	1454 (1344) 58 1547
sprintf_flt (s, "%e", 1.2345)	Flash bytes Stack bytes MCU clocks	3228 (3116) 67 2573	2990 (2880) 67 2311	2968 (2858) 67 2311
sscanf_min ("12345", "%d", &i)	Flash bytes Stack bytes MCU clocks	1532 (1420) 55 1607	1328 (1218) 55 1446	1328 (1218) 55 1446
sscanf ("12345", "%d", &i)	Flash bytes Stack bytes MCU clocks	2008 (1896) 55 1610	1748 (1638) 55 1449	1748 (1638) 55 1449
sscanf ("point,color", "%[a-z]", s)	Flash bytes Stack bytes MCU clocks	2008 (1896) 86 3067	1748 (1638) 86 2806	1748 (1638) 86 2806
sscanf_flt ("1.2345", "%e", &x)	Flash bytes Stack bytes MCU clocks	3464 (3352) 71 2497	3086 (2976) 71 2281	3070 (2960) 71 2078
strtod ("1.2345", &p)	Flash bytes Stack bytes MCU clocks	1632 (1520) 20 1235	1536 (1426) 20 1177	1480 (1480) 21 1124
strtol ("12345", &p, 0)	Flash bytes Stack bytes MCU clocks	918 (806) 22 956	834 (724) 22 891	792 (792) 28 794

9.2 Math functions.

The table contains the number of MCU clocks to calculate a function with a given argument(s). The main reason of a big difference between Avr2 and Avr4 is a hardware multiplication.

Function	Avr2	Avr4
__addsf3 (1.234, 5.678)	113	108
__mulsf3 (1.234, 5.678)	375	138
__divsf3 (1.234, 5.678)	466	465
acos (0.54321)	4411	2455
asin (0.54321)	4517	2556
atan (0.54321)	4710	2271
atan2 (1.234, 5.678)	5270	2857
cbrt (1.2345)	2684	2555
ceil (1.2345)	177	177
cos (1.2345)	3387	1671
cosh (1.2345)	4922	2979
exp (1.2345)	4708	2765
fdim (5.678, 1.234)	111	111
floor (1.2345)	180	180
fmax (1.234, 5.678)	39	37
fmin (1.234, 5.678)	35	35
fmod (5.678, 1.234)	131	131
frexp (1.2345, 0)	42	41
hypot (1.234, 5.678)	1341	866
ldexp (1.2345, 6)	42	42
log (1.2345)	4142	2134
log10 (1.2345)	4498	2260
modf (1.2345, 0)	433	429
pow (1.234, 5.678)	9293	5047
round (1.2345)	150	150

sin (1.2345)	3353	1653
sinh (1.2345)	4946	3003
sqrt (1.2345)	494	492
tan (1.2345)	4381	2426
tanh (1.2345)	5126	3173
trunc (1.2345)	178	178

10 Porting From IAR to AVR GCC

10.1 Introduction

C language was designed to be a portable language. There two main types of porting activities: porting an application to a different platform (OS and/or processor), and porting to a different compiler. Porting to a different compiler can be exacerbated when the application is an embedded system. For example, the C language Standard, strangely, does not specify a standard for declaring and defining Interrupt Service Routines (ISRs). Different compilers have different ways of defining registers, some of which use non-standard language constructs.

This chapter describes some methods and pointers on porting an AVR application built with the IAR compiler to the GNU toolchain (AVR GCC). Note that this may not be an exhaustive list.

10.2 Registers

IO header files contain identifiers for all the register names and bit names for a particular processor. IAR has individual header files for each processor and they must be included when registers are being used in the code. For example:

```
#include <iom169.h>
```

Note

IAR does not always use the same register names or bit names that are used in the AVR datasheet.

AVR GCC also has individual IO header files for each processor. However, the actual processor type is specified as a command line flag to the compiler. (Using the `-mmcu=`*processor* flag.) This is usually done in the Makefile. This allows you to specify only a single header file for any processor type:

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

Note

The forward slash in the <avr/io.h> file name that is used to separate subdirectories can be used on Windows distributions of the toolchain and is the recommended method of including this file.

The compiler knows the processor type and through the single header file above, it can pull in and include the correct individual IO header file. This has the advantage that you only have to specify one generic header file, and you can easily port your application to another processor type without having to change every file to include the new IO header file.

The AVR toolchain tries to adhere to the exact names of the registers and names of the bits found in the AVR datasheet. There may be some descrepencies between the register names found in the IAR IO header files and the AVR GCC IO header files.

10.3 Interrupt Service Routines (ISRs)

As mentioned above, the C language Standard, strangely, does not specify a standard way of declaring and defining an ISR. Hence, every compiler seems to have their own special way of doing so.

IAR declares an ISR like so:

```
#pragma vector=TIMER0_OVF_vect
__interrupt void MotorPWMBottom()
{
    // code
}
```

In AVR GCC, you declare an ISR like so:

```
ISR(PCINT1_vect)
{
    //code
}
```

AVR GCC uses the `ISR` macro to define an ISR. This macro requries the header file:

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

The names of the various interrupt vectors are found in the individual processor IO header files that you must include with `<avr/io.h>`.

Note

The names of the interrupt vectors in AVR GCC has been changed to match the names of the vectors in IAR. This significantly helps in porting applications from IAR to AVR GCC.

10.4 Intrinsic Routines

IAR has a number of intrinsic routine such as

`__enable_interrupts()` `__disable_interrupts()` `__watchdog_reset()`

These intrinsic functions compile to specific AVR opcodes (SEI, CLI, WDR).

There are equivalent macros that are used in AVR GCC, however they are not located in a single include file.

AVR GCC has `sei()` for `__enable_interrupts()`, and `cli()` for `__disable_interrupts()`. Both of these macros are located in `<avr/interrupt.h>`.

AVR GCC has the macro `wdt_reset()` in place of `__watchdog_reset()`. However, there is a whole Watchdog Timer API available in AVR GCC that can be found in `<avr/wdt.h>`.

10.5 Flash Variables

The C language was not designed for Harvard architecture processors with separate memory spaces. This means that there are various non-standard ways to define a variable whose data resides in the Program Memory (Flash).

IAR uses a non-standard keyword to declare a variable in Program Memory:

```
__flash int mydata[] = ....
```

AVR GCC uses Variable Attributes to achieve the same effect:

```
int mydata[] __attribute__((progmem))
```

Note

See the GCC User Manual for more information about Variable Attributes.

AVR-LibC provides a convenience macro for the Variable Attribute:

```
#include <avr/pgmspace.h>
.
.
.
int mydata[] PROGMEM = ....
```

Note

The PROGMEM macro expands to the Variable Attribute of `progmem`. This macro requires that you include <`avr/pgmspace.h`>. This is the canonical method for defining a variable in Program Space.

To read back flash data, use the `pgm_read_*`() macros defined in <`avr/pgmspace.h`>. All Program Memory handling macros are defined there.

There is also a way to create a method to define variables in Program Memory that is common between the two compilers (IAR and AVR GCC). Create a header file that has these definitions:

```
#if defined(__ICCAVR__) // IAR C Compiler
#define FLASH_DECLARE(x) __flash x
#endif
#if defined(__GNUC__) // GNU Compiler
#define FLASH_DECLARE(x) x __attribute__((__progmem__))
#endif
```

This code snippet checks for the IAR compiler or for the GCC compiler and defines a macro `FLASH_DECLARE(x)` that will declare a variable in Program Memory using the appropriate method based on the compiler that is being used. Then you would used it like so:

```
FLASH_DECLARE(int mydata[] = ...);
```

10.6 Non-Returning main()

To declare main() to be a non-returning function in IAR, it is done like this:

```
__C_task void main(void)
{
    // code
}
```

To do the equivalent in AVR GCC, do this:

```
void main(void) __attribute__((noreturn));

void main(void)
{
    //...
}
```

Note

See the GCC User Manual for more information on Function Attributes.

In AVR GCC, a prototype for main() is required so you can declare the function attribute to specify that the main() function is of type "noreturn". Then, define main() as normal. Note that the return type for main() is now `void`.

10.7 Locking Registers

The IAR compiler allows a user to lock general registers from r15 and down by using compiler options and this keyword syntax:

```
__regvar __no_init volatile unsigned int filteredTimeSinceCommutation @14;
```

This line locks r14 for use only when explicitly referenced in your code thorugh the var name "filteredTimeSince↩Commutation". This means that the compiler cannot dispose of it at its own will.

To do this in AVR GCC, do this:

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

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

Note

Do not reserve r0 or r1 as these are used internally by the compiler for a temporary register and for a zero value.

Locking registers is not recommended in AVR GCC as it removes this register from the control of the compiler, which may make code generation worse. Use at your own risk.

11 Frequently Asked Questions

11.1 FAQ Index

- **Interrupts**
  - Why doesn't my program recognize a variable updated in an interrupt routine?
  - Why do some 16-bit timer registers sometimes get trashed?
  - What ISR names are available for my device?
  - What pitfalls exist when writing reentrant code?
  - Why are interrupts re-enabled in the middle of writing the stack pointer?
  - Why are (many) interrupt flags cleared by writing a logical 1?
- **C/C++**
  - Can I use C++ on the AVR?
  - Which -O flag to use?
  - Shouldn't I initialize all my variables?
  - How do I pass an IO port as a parameter to a function?
  - Why do all my "foo...bar" strings eat up the SRAM?
  - How do I put an array of strings completely in ROM?
  - How to modify MCUCR or WDTCR early?
  - How do I perform a software reset of the AVR?
  - On a device with more than 128 KiB of flash, how to make function pointers work?
  - What registers are used by the C compiler?
  - How to permanently bind a variable to a register?
  - Why is assigning ports in a "chain" a bad idea?
  - What is all this _BV() stuff about?
  - Is it really impossible to program the ATtinyXX in C?

11.2 Why doesn't my program recognize a variable updated in an interrupt routine?

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

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

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

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

```
volatile uint8_t flag;
```

Back to FAQ Index.

11.3 How to permanently bind a variable to a register?

This can be done with

```
register uint8_t counter __asm("r3");
```

Typically, it should be safe to use r2 through r7 that way.

Registers r8 through r25 can be used for argument passing by the compiler in case many or long arguments are being passed to callees. If this is not the case throughout the entire application, these registers could be used for register variables as well.

Extreme care should be taken that the entire application is compiled with a consistent set of register-allocated variables including possibly used library functions. This can be achieved by compiling each module with `-ffixed-r3` or `-ffixed-3`. Notice that when you are using library functions from libgcc (the avr-gcc runtime library) or AVR-LibC, then these libraries were generated *without* the requirement to avoid specific registers. Hence when you are using libraries from the distribution, you must make sure that none of the reserved registers is used in the generated binary.

Also notice that global register variables can't be volatile, because only variables in memory can be volatile, and register variables are not located in memory.

Back to FAQ Index.

11.4 How to modify MCUCR or WDTCR early?

Basically, write a small function which looks like this:

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

static __attribute__((used, unused, naked, section(".init3")))
void init_MCUCR (void);

void init_MCUCR (void)
{
    MCUCR = _BV(SRE) | _BV(SRW);
}
```

Do not call this function by hand! This piece of code will be inserted in startup code, which is run right after reset. For the meaning of the attributes, see How do I perform a software reset of the AVR?

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

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

For more information on using sections, see Memory Sections. There is also an example for In C/C++ Code. Note that in C code, any such function would preferably be placed into section `.init3` as the code in `.init2` ensures the internal register `__zero_reg__` is already cleared.

Back to FAQ Index.

11.5 What is all this _BV() stuff about?

When performing low-level output work, which is a very central point in microcontroller programming, it is quite common that a particular bit needs to be set or cleared in some IO register. While the device documentation provides mnemonic names for the various bits in the IO registers, and the AVR device-specific IO definitions reflect these names in definitions for numerical constants, a way is needed to convert a bit number (usually within a byte register) into a byte value that can be assigned directly to the register. However, sometimes the direct bit numbers are needed as well (e. g. in an `SBI()` instruction), so the definitions cannot usefully be made as byte values in the first place.

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

```
_BV(3) => 1 « 3 => 0x08
```

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

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

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

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

Back to FAQ Index.

11.6 Can I use C++ on the AVR?

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

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

- Obviously, none of the C++ related standard functions, classes, and template classes are available.
- The operators `new` and `delete` are not implemented, attempting to use them will cause the linker to complain about undefined external references. (This could perhaps be fixed.)
- Some of the supplied include files are not C++ safe, i. e. they need to be wrapped into

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

  (This could certainly be fixed, too.)
- Exceptions are not supported. Since exceptions are enabled by default in the C++ frontend, they explicitly need to be turned off using `-fno-exceptions` in the compiler options. Failing this, the linker will complain about an undefined external reference to `__gxx_personality_sj0`.

Constructors and destructors *are* supported though, including global ones.

When programming C++ in space- and runtime-sensitive environments like microcontrollers, extra care should be taken to avoid unwanted side effects of the C++ calling conventions like implied copy constructors that could be called upon function invocation etc. These things could easily add up into a considerable amount of time and program memory wasted. Thus, casual inspection of the generated assembler code (using the `-S` compiler option) seems to be warranted.

Back to FAQ Index.

11.7 Shouldn't I initialize all my variables?

Variables in static storage are guaranteed to be initialized by the C standard. This includes global and static variables without explicit initializer, which are initialized to 0. `avr-gcc` does this by placing the appropriate code into section `.init4`. With respect to the standard, this sentence is somewhat simplified (because the standard allows for machines where the actual bit pattern used differs from all bits being 0), but for the AVR target, in general, all integer-type variables are set to 0, all pointers to a NULL pointer, and all floating-point variables to 0.0.

As long as these variables are not explicitly initialized, or their initializer is all zeros, they go into the .bss output section. This section simply records the size of the variable, but otherwise doesn't consume space, neither within the object file nor within flash memory. (Of course, being a variable, it will consume space in the target's SRAM.)

In contrast, global and static variables that have a non-zero initializer go into the .data output section of the file. This will cause them to consume space in the object file (in order to record the initializing value), *and* in the flash ROM of the target device. The latter is needed to initialize the variables in RAM from the initializers kept in ROM during the startup code, so that all variables will have their expected initial values when `main()` is entered.

Back to FAQ Index.

11.8 Why do some 16-bit timer registers sometimes get trashed?

Some of the timer-related 16-bit IO registers use a temporary register (called TEMP in the AVR datasheet) to guarantee an atomic access to the register despite the fact that two separate 8-bit IO transfers are required to actually move the data. Typically, this includes access to the current timer/counter value register (`TCNT`*n*), the input capture register (`ICR`*n*), and write access to the output compare registers (`OCR`*nM*). Refer to the actual datasheet for each device's set of registers that involves the TEMP register.

When accessing one of the registers that use TEMP from the main application, and possibly any other one from within an interrupt routine, care must be taken that no access from within an interrupt context could clobber the TEMP register data of an in-progress transaction that has just started elsewhere.

To protect interrupt routines against other interrupt routines, it's usually best to use the ISR() macro when declaring the interrupt function, and to ensure that interrupts are still disabled when accessing those 16-bit timer registers.

Within the main program, access to those registers could be encapsulated in calls to the cli() and sei() macros. If the status of the global interrupt flag before accessing one of those registers is uncertain, something like the following example code can be used.

```
uint16_t
read_timer1(void)
{
    uint8_t sreg;
    uint16_t val;

    sreg = SREG;
    cli();
    val = TCNT1;
    SREG = sreg;

    return val;
}
```

Back to FAQ Index.

11.9 How do I use a #define'd constant in an asm statement?

So you tried this:
```
asm volatile ("sbi 0x18, 7");
```

Which works. When you do the same thing but replace the address of the port by its macro name, like this:
```
asm volatile ("sbi PORTB, 7");
```

you get a syntax error from the assembler: "`Error: constant value required`".

`PORTB` is a precompiler definition included in the processor specific file included in `avr/io.h`. As you may know, the precompiler will not touch strings, and `PORTB` gets passed to the assembler instead of `0x18`. One way to avoid this problem is:
```
asm volatile ("sbi %0, 7" :: "I" (_SFR_IO_ADDR(PORTB)));
```

Note

For C programs, rather use the standard C bit operators instead, so the above would be expressed as `PORTB |= (1 << 7)`. The optimizer will take care to transform this into a single SBI instruction, assuming the operands allow for this.

There are situation though where the address of a special function register (SFR) is required in inline assembly. When the register can be accessed by `LDS` and `STS`, one can use the RAM address of the SFR:
```
asm volatile ("sts %0, __zero_reg__" :: "n" (& PORTB));
```

When the I/O address of the register is required, one way is to use `_SFR_IO_ADDR` to get the I/O address like in the example above. A different approach is to use inline asm print modifier `i` supported since avr-gcc v4.7:
```
asm volatile ("out %i0, __zero_reg__" :: "n" (& PORTB));
```

The `i0` will print the address of `PORTB` as an I/O address.

Back to FAQ Index.

11.10 Why does the PC randomly jump around when single-stepping through my program in avr-gdb?

When compiling a program with both optimization (`-O`) and debug information (`-g`) which is fortunately possible in `avr-gcc`, the code watched in the debugger is optimized code. It is guaranteed that the code runs with the exact same optimizations as it would run without the `-g` switch.

Since the compiler is free to reorder code execution as long as the semantics do not change, code is often rearranged in order to make it possible to use a single branch instruction for conditional operations. Branch instructions can only cover a short range for the target PC (-63 through +64 words from the current PC). If a branch instruction cannot be used directly, the compiler needs to work around it by combining a skip instruction together with a relative jump (`rjmp`) instruction, which will need one additional word of ROM.

Another side effect of optimization is that variable usage is restricted to the area of code where it is actually used. So if a variable was placed in a register at the beginning of some function, this same register can be re-used later on if the compiler notices that the first variable is no longer used inside that function, even though the variable is still in lexical scope. When trying to examine the variable in `avr-gdb`, the displayed result will then look garbled.

So in order to avoid these side effects, optimization can be turned off while debugging, or at least optimization level `-Og` can be used which was introduced to improve good debugging experience while it still provides a reasonable amount of optimization.

However, some of these optimizations might also have the side effect of uncovering bugs that would otherwise not be obvious, so it must be noted that turning off optimization can easily change the bug pattern. In most cases, you are better off leaving optimizations enabled while debugging.

Back to FAQ Index.

11.11 How do I trace an assembler file in avr-gdb?

When using the `-g` compiler option, `avr-gcc` only generates line number and other debug information for C (and C++) files that pass the compiler. Functions that don't have line number information will be completely skipped by a single `step` command in `gdb`. This includes functions linked from a standard library, but by default also functions defined in an assembler source file, since the `-g` compiler switch does not apply to the assembler.

So in order to debug an assembler input file (possibly one that has to be passed through the C preprocessor), it's the assembler that needs to be told to include line-number information into the output file. (Other debug information like data types and variable allocation cannot be generated, since unlike a compiler, the assembler basically doesn't know about this.) This is done using the (GNU) assembler option `--gstabs`.

Example:

```
$ avr-as -mmcu=atmega128 --gstabs -o foo.o foo.s
```

When the assembler is not called directly but through the C compiler frontend (either implicitly by passing a source file ending in `.S`, or explicitly using `-x assembler-with-cpp`), the compiler frontend needs to be told to pass the `--gstabs` option down to the assembler. This is done using `-Wa,--gstabs`. Please take care to *only* pass this option when compiling an assembler input file. Otherwise, the assembler code that results from the C compilation stage will also get line number information, which confuses the debugger.

Note

You can also use `-Wa,-gstabs` since the compiler will add the extra `'-'` for you.

Example:

```
$ EXTRA_OPTS="-Wall -mmcu=atmega128 -x assembler-with-cpp"
$ avr-gcc -Wa,--gstabs ${EXTRA_OPTS} -c -o foo.o foo.S
```

Also note that the debugger might get confused when entering a piece of code that has a non-local label before, since it then takes this label as the name of a new function that appears to have been entered. Thus, the best practice to avoid this confusion is to only use non-local labels when declaring a new function, and restrict anything else to local labels. Local labels consist just of a number only. References to these labels consist of the number, followed by the letter **b** for a backward reference, or **f** for a forward reference. These local labels may be re-used within the source file, references will pick the closest label with the same number and given direction.

Example:

```
myfunc:
    push    r16
    push    r17
    push    r18
    push    YL
    push    YH
    ...
    clr     r16             ; start loop
    ldi     YL, lo8(sometable)
    ldi     YH, hi8(sometable)
    rjmp    2f              ; jump to loop test at end
1:  ld      r17, Y+         ; loop continues here
    ...
    breq    3f              ; return from myfunc prematurely
    ...
    inc     r16
2:  cmp     r16, r18
    brlo    1b              ; jump back to top of loop
3:  pop     YH
    pop     YL
    pop     r18
    pop     r17
    pop     r16
    ret
```

Back to FAQ Index.

11.12 How do I pass an IO port as a parameter to a function?

Consider this example code:

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

void
set_bits_func_wrong (volatile uint8_t port, uint8_t mask)
{
    port |= mask;
}

void
set_bits_func_correct (volatile uint8_t *port, uint8_t mask)
{
    *port |= mask;
}

#define set_bits_macro(port,mask) ((port) |= (mask))

int main (void)
{
    set_bits_func_wrong (PORTB, 0xaa);
    set_bits_func_correct (&PORTB, 0x55);
    set_bits_macro (PORTB, 0xf0);

    return (0);
}
```

The first function will generate object code which is not even close to what is intended. The major problem arises when the function is called. When the compiler sees this call, it will actually pass the value of the `PORTB` register (using an `IN` instruction), instead of passing the address of `PORTB` (e.g. memory mapped io addr of `0x38`, io port `0x18` for the mega128). This is seen clearly when looking at the disassembly of the call:

```
    set_bits_func_wrong (PORTB, 0xaa);
 10a:   6a ea           ldi     r22, 0xAA
 10c:   88 b3           in      r24, 0x18
 10e:   0e 94 65 00     call    0xca
```

So, the function, once called, only sees the value of the port register and knows nothing about which port it came from. At this point, whatever object code is generated for the function by the compiler is irrelevant. The interested reader can examine the full disassembly to see that the function's body is completely fubar.

The second function shows how to pass (by reference) the memory mapped address of the io port to the function so that you can read and write to it in the function. Here's the object code generated for the function call:

```
    set_bits_func_correct (&PORTB, 0x55);
 112:   65 e5           ldi     r22, 0x55
 114:   88 e3           ldi     r24, 0x38
 116:   90 e0           ldi     r25, 0x00
 118:   0e 94 7c 00     call    0xf8
```

You can clearly see that `0x0038` is correctly passed for the address of the io port. Looking at the disassembled object code for the body of the function, we can see that the function is indeed performing the operation we intended:

```
void
set_bits_func_correct (volatile uint8_t *port, uint8_t mask)
{
  f8:   fc 01           movw    r30, r24
    *port |= mask;
  fa:   80 81           ld      r24, Z
  fc:   86 2b           or      r24, r22
  fe:   80 83           st      Z, r24
}
 100:   08 95           ret
```

Notice that we are accessing the io port via the `LD` and `ST` instructions.

The `port` parameter must be volatile to avoid a compiler warning.

Note

Because of the nature of the IN and OUT assembly instructions, they can not be used inside the function when passing the port in this way. Readers interested in the details should consult the *Instruction Set* datasheet.

Finally we come to the macro version of the operation. In this contrived example, the macro is the most efficient method with respect to both execution speed and code size:

```
    set_bits_macro (PORTB, 0xf0);
 11c:    88 b3           in      r24, 0x18
 11e:    80 6f           ori     r24, 0xF0
 120:    88 bb           out     0x18, r24
```

Of course, in a real application, you might be doing a lot more in your function which uses a passed by reference io port address and thus the use of a function over a macro could save you some code space, but still at a cost of execution speed.

Care should be taken when such an indirect port access is going to one of the 16-bit IO registers where the order of write access is critical (like some timer registers). All versions of avr-gcc up to 3.3 will generate instructions that use the wrong access order in this situation (since with normal memory operands where the order doesn't matter, this sometimes yields shorter code).

See http://mail.gnu.org/archive/html/avr-libc-dev/2003-01/msg00044.html for a possible workaround.

avr-gcc versions after 3.3 have been fixed in a way where this optimization will be disabled if the respective pointer variable is declared to be volatile, so the correct behaviour for 16-bit IO ports can be forced that way.

Back to FAQ Index.

11.13 What registers are used by the C compiler?

See also the Type Layout, Register Layout and Calling Convention sections in the avr-gcc Wiki.

Data types char is 8 bits, int and short are 16 bits, long is 32 bits, long long is 64 bits, float is 32 bits, double and long double are 32 bits or 64 bits, pointers are 16 bits (function pointers are word addresses to allow addressing up to 128K program memory space).

- There is a -mint8 option (see Options for the C compiler avr-gcc) to make int and short 8 bits, long 16 bits and long long 32 bits. But that is not supported by AVR-LibC (except for stdint.h and avr/pgmspace.h, but no 64-bit integer types are available) and violates C standards (int *must* be at least 16 bits).

Call-used registers (r18-r27, r30-r31) May be allocated by gcc for local data. You *may* use them freely in assembly subroutines. Calling C subroutines can clobber any of them - the caller is responsible for saving and restoring.

For the AVR_TINY architecture (ATtiny10 and relatives), r20-r27 and r30-31 are call-clobbered.

Call-saved registers (r2-r17, r28-r29) May be allocated by gcc for local data. Calling C subroutines leaves them unchanged. Assembly subroutines are responsible for saving and restoring these registers, if changed. r29↔:r28 (Y pointer) is used as a frame pointer (points to local data on stack) if necessary. The requirement for the callee to save/preserve the contents of these registers even applies in situations where the compiler assigns them for argument passing.

For the AVR_TINY architecture (ATtiny10 etc.), r18-r19 and r28-r29 are call-saved. Registers r0 through r15 do not exist.

Fixed registers (r0, r1) Never allocated by gcc for local data, but often used for fixed purposes:

- **r0** (`__tmp_reg__`) — temporary register, can be clobbered by any code (except interrupt handlers which save it), *may* be used to remember something for a while within one piece of assembly code
- **r1** (`__zero_reg__`) — assumed to be always zero in any C code, *may* be used to remember something for a while within one piece of assembler code, but *must* then be cleared after use (`clr __zero_reg__`). This includes any use of the `[f]mul[s[u]]` instructions, which return their result in r1:r0. Interrupt handlers save and clear `__zero_reg__` on entry, and restore it on exit (in case it was non-zero).
- **T flag** — the `T` flag in the status register (`SREG`) can be used the same way like `__tmp_reg__`.

For the AVR_TINY architecture (ATtiny10 etc.), `__tmp_reg__` is r16, and `__zero_reg__` is r17.

Function call conventions Arguments - allocated left to right, r25 to r8. All arguments are aligned to start in even-numbered registers (odd-sized arguments, including `char`, have one free register above them). This allows making better use of the `movw` instruction on the enhanced core.

If too many, those that don't fit are passed on the stack.

On AVR_TINY, r25 to r20 are used to pass values.

- Return values: 8-bit in r24, 16-bit in r25:r24, up to 32 bits in r22-r25, up to 64 bits in r18-r25.
- Arguments to functions with a variable number of lists like `printf` get all their values on the stack. `char` is extended to `int`, and `float` is extended to `double`.
- When an argument is passed on the stack, all subsequent arguments are also passed on the stack.
- An argument is either passed completely in registers or completely on the stack.
- Arguments with a size of zero or with a size larger than 8 bytes (4 bytes on AVR_TINY) are returned in memory. The caller provides the memory location as implicit first argument to the callee.
- When an argument is returned in registers, its size is padded to the next power of 2.

Back to FAQ Index.

11.14 How do I put an array of strings completely in ROM?

There are times when you may need an array of strings which will never be modified. In this case, you don't want to waste ram storing the constant strings. The most obvious (and incorrect) thing to do is this:

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

const char* const array[2] PROGMEM = {
    "Foo",
    "Bar"
};

int main (void)
{
    char buf[32];
    strcpy_P (buf, array[1]);
    return 0;
}
```

The result is not what you want though. What you end up with is the array stored in ROM, while the individual strings end up in RAM (in some `.rodata` input section).

To work around this, you need to do something like this:

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

static const char foo[] PROGMEM = "Foo";
static const char bar[] PROGMEM = "Bar";

const char* const array[2] PROGMEM = {
    foo,
    bar
```

```
};

void func (uint8_t i)
{
    char buf[32];

    const char *p = pgm_read_ptr (&array[i]);
    strcpy_P (buf, p);
}
```

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

```
00000026 <array>:
  26:   2e 00 2a 00

0000002a <bar>:
  2a:   42 61 72 00                                          Bar.

0000002e <foo>:
  2e:   46 6f 6f 00                                          Foo.
```

`foo` is at address 0x002e.
`bar` is at address 0x002a.
`array` is at address 0x0026.

11.14.1 Using named address-spaces

An alternative is to use the named address-space `__flash`, which is supported since avr-gcc v4.7 and in GNU-C99 and up:

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

#define F(X) ((const __flash char[]) { X })

const __flash char* const __flash array[] =
{
    F("Foo"), F("Bar")
};

int compare (const char *str, uint8_t i)
{
    return strcmp_P (str, array[i]);
}
```

Moreover, there is no more need for `pgm_read_xxx()`: The (addresses of) the string literals can be read directly by means of `array[i]`. The header is only needed for the `strcmp_P` prototype.

Back to FAQ Index.

11.15 How to use external RAM?

Well, there is no universal answer to this question; it depends on what the external RAM is going to be used for.

Basically, the bit `SRE` (SRAM enable) in the `MCUCR` register needs to be set in order to enable the external memory interface. Depending on the device to be used, and the application details, further registers affecting the external memory operation like `XMCRA` and `XMCRB`, and/or further bits in `MCUCR` might be configured. Refer to the datasheet for details.

If the external RAM is going to be used to store the variables from the C program (i. e., the `.data` and/or `.bss` segment) in that memory area, it is essential to set up the external memory interface early during the device initialization so the initialization of these variable will take place. Refer to How to modify MCUCR or WDTCR early? for a description how to do this using few lines of assembler code, or to the chapter about memory sections for an example written in C.

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

Finally, if the application simply wants to use the additional RAM for private data storage kept outside the domain of the C compiler (e. g. through a `char *` variable initialized directly to a particular address), it would be sufficient to defer the initialization of the external RAM interface to the beginning of **`main()`**, so no tweaking of the `.init3` section is necessary. The same applies if only the heap is going to be located there, since the application start-up code does not affect the heap.

It is not recommended to locate the stack in external RAM. In general, accessing external RAM is slower than internal RAM, and errata of some AVR devices even prevent this configuration from working properly at all.

Back to FAQ Index.

11.16 Which -O flag to use?

There's a common misconception that larger numbers behind the `-O` option might automatically cause "better" optimization. First, there's no universal definition for "better", with optimization often being a speed vs. code size trade off. See the detailed discussion for which option affects which part of the code generation.

A test case was run on an ATmega128 to judge the effect of compiling the library itself using different optimization levels. The following table lists the results. The test case consisted of around 2 KB of strings to sort. Test #1 used qsort() using the standard library strcmp(), test #2 used a function that sorted the strings by their size (thus had two calls to strlen() per invocation).

When comparing the resulting code size, it should be noted that a floating point version of fvprintf() was linked into the binary (in order to print out the time elapsed) which is entirely not affected by the different optimization levels, and added about 2.5 KB to the code.

Optimization Flags	Size of .text	Time for Test #1	Time for Test #2
-O3	6898	903 μs	19.7 ms
-O2	6666	972 μs	20.1 ms
-Os	6618	955 μs	20.1 ms
-Os -mcall-prologues	6474	972 μs	20.1 ms

(The difference between 955 μs and 972 μs was just a single timer-tick, so take this with a grain of salt.)

So generally, it seems `-Os -mcall-prologues` is the most universal "best" optimization level. Only applications that need to get the last few percent of speed benefit from using `-O3`.

Back to FAQ Index.

11.17 How do I relocate code to a fixed address?

First, put the function into a new, orphan named section. This is done with a `section` attribute that specifies the name of the input section with the prototype of the function:

```
__attribute__ ((noinline, noclone, section (".bootloader")))
void boot (void);
```

The `noinline` and `noclone` attributes are required to make sure that the function is not (partially) inlined into the caller, which does not have a respective section attribute.

Second, locate the section to the desired fixed address by means of linking with, say

```
-Wl,--section-start,.bootloader=0x1E000
```

see the -Wl compiler option. The name after `--section-start` is the name of the section to be located, and the number specifies the beginning address of the named section.

This will only work when the section is an orphan section, i.e. a section that is not mentioned in the linker script. For sections that *are* mentioned in the linker script, like for example `.text.bootloader`, this will not work because `--section-start` refers to an output section, but the output section for input section `.text.bootloader` is the `.text` section.

To verify that everything went as expected, generate the disassembly with `avr-objdump ... -j .bootloader`. The top of the list file will show

```
1 .bootloader   00000004  00002000  00002000  00000204  2**0
                CONTENTS, ALLOC, LOAD, READONLY, CODE
```

Or display section properties with `avr-readelf --section-details`

```
$ avr-readelf -t main.elf
Section Headers:
  [Nr] Name
       Type            Addr     Off    Size   ES   Lk Inf Al
       Flags
  [ 2] .bootloader
       PROGBITS        00002000 000204 000004 00   0   0  1
       [00000006]: ALLOC, EXEC
```

A different way to locate the section is by means of a custom linker script. The `avr-gcc Wiki` has an example that locates the `.progmem2.data` section which is used by the compiler for variables in address-space `__flash2`.

Back to FAQ Index.

11.18 My UART is generating nonsense! My ATmega128 keeps crashing! Port F is completely broken!

Well, certain odd problems arise out of the situation that the AVR devices as shipped by Atmel often come with a default fuse bit configuration that doesn't match the user's expectations. Here is a list of things to care for:

- All devices that have an internal RC oscillator ship with the fuse enabled that causes the device to run off this oscillator, instead of an external crystal. This often remains unnoticed until the first attempt is made to use something critical in timing, like UART communication.
- The ATmega128 ships with the fuse enabled that turns this device into ATmega103 compatibility mode. This means that some ports are not fully usable, and in particular that the internal SRAM is located at lower addresses. Since by default, the stack is located at the top of internal SRAM, a program compiled for an ATmega128 running on such a device will immediately crash upon the first function call (or rather, upon the first function return).
- Devices with a JTAG interface have the `JTAGEN` fuse programmed by default. This will make the respective port pins that are used for the JTAG interface unavailable for regular IO.

Back to FAQ Index.

11.19 Why do all my "foo...bar" strings eat up the SRAM?

By default, all strings are handled as all other initialized variables: they occupy RAM (even though the compiler might warn you when it detects write attempts to these RAM locations), and occupy the same amount of flash ROM so they can be initialized to the actual string by startup code.

That way, any string literal will be a valid argument to any C function that expects a `const char*` argument.

Of course, this is going to waste a lot of SRAM. In Program Space String Utilities, a method is described how such constant data can be moved out to flash ROM. However, a constant string located in flash ROM is no longer a valid argument to pass to a function that expects a `const char*`-type string, since the AVR processor needs the special instruction `LPM` to access these strings. Thus, separate functions are needed that take this into account. Many of the standard C library functions have equivalents available where one of the string arguments can be located in flash ROM. Private functions in the applications need to handle this, too. For example, the following can be used to implement simple debugging messages that will be sent through a UART:

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

int uart_putchar(char c)
{
  if (c == '\n')
    uart_putchar('\r');
  loop_until_bit_is_set(USR, UDRE);
  UDR = c;
  return 0; /* so it could be used for fdevopen(), too */
}

void debug_P(const char *addr)
{
  char c;

  while ((c = pgm_read_byte(addr++)))
    uart_putchar(c);
}

int main(void)
{
  ioinit(); /* initialize UART, ... */
  debug_P(PSTR("foo was here\n"));
  return 0;
}
```

Note

By convention, the suffix **_P** to the function name is used as an indication that this function is going to accept a "program-space string". Note also the use of the PSTR() macro.

Back to FAQ Index.

11.20 How to detect RAM memory and variable overlap problems?

You can simply run `avr-nm` on your output (ELF) file. Run it with the `-n` option, and it will sort the symbols numerically (by default, they are sorted alphabetically).

Look for the symbol `_end`, that's the first address in RAM that is not allocated by a variable. (avr-gcc internally adds 0x800000 to all data/bss variable addresses, so please ignore this offset.) Then, the run-time initialization code initializes the stack pointer (by default) to point to the last available address in (internal) SRAM. Thus, the region between `_end` and the end of SRAM is what is available for stack. (If your application uses malloc(), which e. g. also can happen inside printf(), the heap for dynamic memory is also located there. See Memory Areas and Using malloc().)

The amount of stack required for your application cannot be determined that easily. For example, if you recursively call a function and forget to break that recursion, the amount of stack required is infinite. :-) You can look at the generated assembler code (`avr-gcc ... -S`), there's a comment in each generated assembler file that tells you the frame size for each generated function. That's the amount of stack required for this function, you have to add up that for all functions where you know that the calls could be nested.

Back to FAQ Index.

11.21 Is it really impossible to program the ATtinyXX in C?

While some small AVRs are not directly supported by the C compiler since they do not have a RAM-based stack (and some do not even have RAM at all), it is possible anyway to use the general-purpose registers as a RAM replacement since they are mapped into the data memory region.

Bruce D. Lightner wrote an excellent description of how to do this, and offers this together with a toolkit on his web page:

`http://lightner.net/avr/ATtinyAvrGcc.html`

Back to FAQ Index.

11.22 What is this "clock skew detected" message?

It's a known problem of the MS-DOS FAT file system. Since the FAT file system has only a granularity of 2 seconds for maintaining a file's timestamp, and it seems that some MS-DOS derivative (Win9x) perhaps rounds up the current time to the next second when calculating the timestamp of an updated file in case the current time cannot be represented in FAT's terms, this causes a situation where `make` sees a "file coming from the future".

Since all make decisions are based on file timestamps, and their dependencies, make warns about this situation.

Solution: don't use inferior file systems / operating systems. Neither Unix file systems nor HPFS (aka NTFS) do experience that problem.

Workaround: after saving the file, wait a second before starting `make`. Or simply ignore the warning. If you are paranoid, execute a `make clean all` to make sure everything gets rebuilt.

In networked environments where the files are accessed from a file server, this message can also happen if the file server's clock differs too much from the network client's clock. In this case, the solution is to use a proper time keeping protocol on both systems, like NTP. As a workaround, synchronize the client's clock frequently with the server's clock.

Back to FAQ Index.

11.23 Why are (many) interrupt flags cleared by writing a logical 1?

Usually, each interrupt has its own interrupt flag bit in some control register, indicating the specified interrupt condition has been met by representing a logical 1 in the respective bit position. When working with interrupt handlers, this interrupt flag bit usually gets cleared automatically in the course of processing the interrupt, sometimes by just calling the handler at all, sometimes (e. g. for the U[S]ART) by reading a particular hardware register that will normally happen anyway when processing the interrupt.

From the hardware's point of view, an interrupt is asserted as long as the respective bit is set, while global interrupts are enabled. Thus, it is essential to have the bit cleared before interrupts get re-enabled again (which usually happens when returning from an interrupt handler).

Only few subsystems require an explicit action to clear the interrupt request when using interrupt handlers. (The notable exception is the TWI interface, where clearing the interrupt indicates to proceed with the TWI bus hardware handshake, so it's never done automatically.)

However, if no normal interrupt handlers are to be used, or in order to make extra sure any pending interrupt gets cleared before re-activating global interrupts (e. g. an external edge-triggered one), it can be necessary to explicitly clear the respective hardware interrupt bit by software. This is usually done by writing a logical 1 into this bit position.

This seems to be illogical at first, the bit position already carries a logical 1 when reading it, so why does writing a logical 1 to it *clear* the interrupt bit?

The solution is simple: writing a logical 1 to it requires only a single OUT instruction, and it is clear that only this single interrupt request bit will be cleared. There is no need to perform a read-modify-write cycle (like, an SBI instruction), since all bits in these control registers are interrupt bits, and writing a logical 0 to the remaining bits (as it is done by the simple OUT instruction) will not alter them, so there is no risk of any race condition that might accidentally clear another interrupt request bit. So instead of writing

```
TIFR |= _BV(TOV0); /* wrong! */
```

simply use

```
TIFR = _BV(TOV0);
```

Back to FAQ Index.

11.24 Why have "programmed" fuses the bit value 0?

Basically, fuses are just a bit in a special EEPROM area. For technical reasons, erased E[E]PROM cells have all bits set to the value 1, so unprogrammed fuses also have a logical 1. Conversely, programmed fuse cells read out as bit value 0.

Back to FAQ Index.

11.25 Which AVR-specific assembler operators are available?

See Pseudo-Ops and Operand Modifiers.

Back to FAQ Index.

11.26 Why are interrupts re-enabled in the middle of writing the stack pointer?

When setting up space for local variables on the stack, the compiler generates code like this:

```
/* prologue: frame size=20 */
    push r28
    push r29
    in r28,__SP_L__
    in r29,__SP_H__
    sbiw r28,20
    in __tmp_reg__,__SREG__
    cli
    out __SP_H__,r29
    out __SREG__,__tmp_reg__
    out __SP_L__,r28
/* prologue end (size=10) */
```

It reads the current stack pointer value, decrements it by the required amount of bytes, then disables interrupts, writes back the high part of the stack pointer, writes back the saved SREG (which will eventually re-enable interrupts if they have been enabled before), and finally writes the low part of the stack pointer.

At the first glance, there's a race between restoring SREG, and writing SPL. However, after enabling interrupts (either explicitly by setting the I flag, or by restoring it as part of the entire SREG), the AVR hardware executes (at least) the next instruction still with interrupts disabled, so the write to SPL is guaranteed to be executed with interrupts disabled still. Thus, the emitted sequence ensures interrupts will be disabled only for the minimum time required to guarantee the integrity of this operation.

Back to FAQ Index.

11.27 Why are there five different linker scripts?

From a comment in the source code:

Which one of the five linker script files is actually used depends on command line options given to ld.

A .x script file is the default script A .xr script is for linking without relocation (-r flag) A .xu script is like .xr but ∗do∗ create constructors (-Ur flag) A .xn script is for linking with -n flag (mix text and data on same page). A .xbn script is for linking with -N flag (mix text and data on same page).

Back to FAQ Index.

11.28 How to add a raw binary image to linker output?

The GNU linker `avr-ld` cannot handle binary data directly. However, there's a companion tool called `avr-objcopy`. This is already known from the output side: it's used to extract the contents of the linked ELF file into an Intel Hex load file.

`avr-objcopy` can create a relocatable object file from arbitrary binary input, like

```
avr-objcopy -I binary -O elf32-avr foo.bin foo.o
```

This will create a file named `foo.o`, with the contents of `foo.bin`. The contents will default to section .data, and two symbols will be created named `_binary_foo_bin_start` and `_binary_foo_bin_end`. These symbols can be referred to inside a C source to access these data.

If the goal is to have those data go to flash ROM (similar to having used the PROGMEM attribute in C source code), the sections have to be renamed while copying, and it's also useful to set the section flags:

```
avr-objcopy --rename-section .data=.progmem.data,contents,alloc,load,readonly,data -I binary -O elf32-avr
      foo.bin foo.o
```

Note that all this could be conveniently wired into a Makefile, so whenever `foo.bin` changes, it will trigger the recreation of `foo.o`, and a subsequent relink of the final ELF file.

Below are two Makefile fragments that provide rules to convert a .txt file to an object file, and to convert a .bin file to an object file:

```
$(OBJDIR)/%.o : %.txt
    @echo Converting $<
    @cp $(<) $(*).tmp
    @echo -n 0 | tr 0 '\000' » $(*).tmp
    @$(OBJCOPY) -I binary -O elf32-avr \
    --rename-section .data=.progmem.data,contents,alloc,load,readonly,data \
    --redefine-sym _binary_$*_tmp_start=$* \
    --redefine-sym _binary_$*_tmp_end=$*_end \
    --redefine-sym _binary_$*_tmp_size=$*_size_sym \
    $(*).tmp $(@)
    @echo "extern const char" $(*)"[] PROGMEM;" > $(*).h
    @echo "extern const char" $(*)_end"[] PROGMEM;" » $(*).h
    @echo "extern const char" $(*)_size_sym"[];" » $(*).h
    @echo "#define $(*)_size ((int)$(*)_size_sym)" » $(*).h
    @rm $(*).tmp

$(OBJDIR)/%.o : %.bin
    @echo Converting $<
    @$(OBJCOPY) -I binary -O elf32-avr \
    --rename-section .data=.progmem.data,contents,alloc,load,readonly,data \
    --redefine-sym _binary_$*_bin_start=$* \
    --redefine-sym _binary_$*_bin_end=$*_end \
    --redefine-sym _binary_$*_bin_size=$*_size_sym \
    $(<) $(@)
    @echo "extern const char" $(*)"[] PROGMEM;" > $(*).h
    @echo "extern const char" $(*)_end"[] PROGMEM;" » $(*).h
    @echo "extern const char" $(*)_size_sym"[];" » $(*).h
    @echo "#define $(*)_size ((int)$(*)_size_sym)" » $(*).h
```

Back to FAQ Index.

11.29 How do I perform a software reset of the AVR?

The canonical way to perform a software reset of non-XMega AVR's is to use the watchdog timer. Enable the watchdog timer to the shortest timeout setting, then go into an infinite, do-nothing loop. The watchdog will then reset the processor.

XMega parts have a specific bit `RST_SWRST_bm` in the `RST.CTRL` register, that generates a hardware reset. RST_SWRST_bm is protected by the XMega Configuration Change Protection system.

The reason why using the watchdog timer or `RST_SWRST_bm` is preferable over jumping to the reset vector, is that when the watchdog or `RST_SWRST_bm` resets the AVR, the registers will be reset to their known, default settings. Whereas jumping to the reset vector will leave the registers in their previous state, which is generally not a good idea.

CAUTION! Older AVRs will have the watchdog timer disabled on a reset. For these older AVRs, doing a soft reset by enabling the watchdog is easy, as the watchdog will then be disabled after the reset. On newer AVRs, once the watchdog is enabled, then it **stays enabled, even after a reset**! For these newer AVRs a function needs to be added to the `.init3` section (i.e. during the startup code, before main()) to disable the watchdog early enough so it does not continually reset the AVR.

Here is some example code that creates a macro that can be called to perform a soft reset:

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

static inline __attribute__((__always_inline__))
void soft_reset (void)
{
    wdt_enable (WDTO_15MS);
    for(;;) {}
}
```

For newer AVRs (such as the ATmega1281) also add this function to your code to then disable the watchdog after a reset (e.g., after a soft reset):

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

// Function Pototype
static  __attribute__((used, unused, naked, section(".init3")))
void wdt_init (void);

// Function Implementation
void wdt_init (void)
{
    MCUSR = 0;
    wdt_disable();
}
```

The code is placed in section `.init3` so that it is executed as part of the normal startup procedure. The `naked` attribute is required so that the code does not `return` (Code in init sections is executed as it is located; the code is not called, and code from one init section falls through to the code in the next one). The `used` attribute makes sure that the compiler does not throw the seemingly unused function away. The `unused` attributes avoids warnings about unused code.

Back to FAQ Index.

11.30 What pitfalls exist when writing reentrant code?

Reentrant code means the ability for a piece of code to be called simultaneously from two or more threads. Attention to re-enterability is needed when using a multi-tasking operating system, or when using interrupts since an interrupt is really a temporary thread.

The code generated natively by gcc is reentrant. But, only some of the libraries in AVR-LibC are explicitly reentrant, and some are known not to be reentrant. In general, any library call that reads and writes global variables (including I/O registers) is not reentrant. This is because more than one thread could read or write the same storage at the same time, unaware that other threads are doing the same, and create inconsistent and/or erroneous results.

A library call that is known not to be reentrant will work if it is used only within one thread *and* no other thread makes use of a library call that shares common storage with it.

Below is a table of library calls with known issues.

Library Call	Reentrant Issue	Workaround / Alternative
rand, random	Uses global variables to keep state information.	Use special reentrant versions↩ : rand_r, random_r.
strtof, strtod, strtol, strtoul	Uses the global variable errno to return success/failure.	Ignore errno, or protect calls with cli/sei or ATOMIC_BLOCK if the application can tolerate it. Or use scanf or scanf_P if possible.
malloc, realloc, calloc, free	Uses the stack pointer and global variables to allocate and free memory.	Protect calls with cli/sei or ATOMIC_BLOCK if the application can tolerate it. If using an OS, use the OS provided memory allocator since the OS is likely modifying the stack pointer anyway.
fdevopen, fclose	Uses calloc and free.	Protect calls with cli/sei or ATOMIC_BLOCK if the application can tolerate it. Or use fdev_setup_stream or FDEV_SETUP_STREAM. Note: fclose will only call free if the stream has been opened with fdevopen.
eeprom_*, boot_*	Accesses I/O registers.	Protect calls with cli/sei, ATOMIC_BLOCK, or use OS locking.
pgm_*_far	Accesses I/O register RAMPZ.	Starting with GCC 4.3, RAMPZ is automatically saved for ISRs, so nothing further is needed if only using interrupts. Some OSes may automatically preserve RAMPZ during context switching. Check the OS documentation before assuming it does. Otherwise, protect calls with cli/sei, ATOMIC_BLOCK, or use explicit OS locking.
printf, printf_P, vprintf, puts, puts_P	Alters flags and character count in global FILE stdout.	Use only in one thread. Or if returned character count is unimportant, do not use the *_P versions. Note: Formatting to a string output, e.g. sprintf, sprintf_P, snprintf, snprintf_P, vsprintf, vsprintf_P, vsnprintf, vsnprintf_P, is thread safe. The formatted string could then be followed by an fwrite which simply calls the lower layer to send the string.
fprintf, fprintf_P, vfprintf, vfprintf_P, fputs, fputs_P	Alters flags and character count in the FILE argument. Problems can occur if a global FILE is used from multiple threads.	Assign each thread its own FILE for output. Or if returned character count is unimportant, do not use the *_P versions.
assert	Contains an embedded fprintf. See above for fprintf.	See above for fprintf.

Library Call	Reentrant Issue	Workaround / Alternative
`clearerr`	Alters flags in the FILE argument.	Assign each thread its own FILE for output.
`getchar`, `gets`	Alters flags, character count, and unget buffer in global FILE `stdin`.	Use only in one thread. ***
`fgetc`, `ungetc`, `fgets`, `scanf`, `scanf_P`, `fscanf`, `fscanf_P`, `vscanf`, `vfscanf`, `vfscanf_P`, `fread`	Alters flags, character count, and unget buffer in the FILE argument.	Assign each thread its own FILE for input. *** Note: Scanning from a string, e.g. sscanf and sscanf_P, are thread safe.

Note

It's not clear one would ever want to do character input simultaneously from more than one thread anyway, but these entries are included for completeness.

An effort will be made to keep this table up to date if any new issues are discovered or introduced.

Back to FAQ Index.

11.31 Why are some addresses of the EEPROM corrupted (usually address zero)?

The two most common reason for EEPROM corruption is either writing to the EEPROM beyond the datasheet endurance specification, or resetting the AVR while an EEPROM write is in progress.

EEPROM writes can take up to tens of milliseconds to complete. So that the CPU is not tied up for that long of time, an internal state-machine handles EEPROM write requests. The EEPROM state-machine expects to have all of the EEPROM registers setup, then an EEPROM write request to start the process. Once the EEPROM state-machine has started, changing EEPROM related registers during an EEPROM write is guaranteed to corrupt the EEPROM write process. The datasheet always shows the proper way to tell when a write is in progress, so that the registers are not changed by the user's program. The EEPROM state-machine will **always** complete the write in progress unless power is removed from the device.

As with all EEPROM technology, if power fails during an EEPROM write the state of the byte being written is undefined.

In older generation AVRs the EEPROM Address Register (EEAR) is initialized to zero on reset, be it from Brown Out Detect, Watchdog or the Reset Pin. If an EEPROM write has just started at the time of the reset, the write will be completed, but now at address zero instead of the requested address. If the reset occurs later in the write process both the requested address and address zero may be corrupted.

To distinguish which AVRs may exhibit the corrupt of address zero while a write is in process during a reset, look at the "initial value" section for the EEPROM Address Register. If EEAR shows the initial value as 0x00 or 0x0000, then address zero and possibly the one being written will be corrupted. Newer parts show the initial value as "undefined", these will not corrupt address zero during a reset (unless it was address zero that was being written).

EEPROMs have limited write endurance. The datasheet specifies the number of EEPROM writes that are guaranteed to function across the full temperature specification of the AVR, for a given byte. A read should always be performed before a write, to see if the value in the EEPROM actually needs to be written, so not to cause unnecessary EEPROM wear.

The failure mechanism for an overwritten byte is generally one of "stuck" bits, i. e. a bit will stay at a one or zero state regardless of the byte written. Also a write followed by a read may return the correct data, but the data will change with the passage of time, due the EEPROM's inability to hold a charge from the excessive write wear.

Back to FAQ Index.

11.32 Why is my baud rate wrong?

Some AVR datasheets give the following formula for calculating baud rates:

```
(F_CPU/(UART_BAUD_RATE*16L)-1)
```

Unfortunately that formula does not work with all combinations of clock speeds and baud rates due to integer truncation during the division operator.

When doing integer division it is usually better to round to the nearest integer, rather than to the lowest. To do this add 0.5 (i. e. half the value of the denominator) to the numerator before the division, resulting in the formula:

```
((F_CPU + UART_BAUD_RATE * 8L) / (UART_BAUD_RATE * 16L) - 1)
```

This is also the way it is implemented in <util/setbaud.h>: Helper macros for baud rate calculations.

Back to FAQ Index.

11.33 On a device with more than 128 KiB of flash, how to make function pointers work?

Function pointers beyond the "magical" 128 KiB barrier(s) on larger devices are supposed to be resolved through so-called *trampolines* by the linker, so the actual pointers used in the code can remain 16 bits wide.

In order for this to work, the option `-mrelax` must be given on the compiler command-line that is used to link the final ELF file. (Older compilers did not implement this option for the AVR, use `-Wl,--relax` instead.)

See also the avr-gcc online documentation on the `EIND special function register` and indirect calls.

Back to FAQ Index.

11.34 Why is assigning ports in a "chain" a bad idea?

Suppose a number of IO port registers should get the value `0xff` assigned. Conveniently, it is implemented like this:

```
DDRB = DDRD = 0xff;
```

According to the rules of the C language, this causes 0xff to be assigned to `DDRD`, then `DDRD` is read back, and the value is assigned to `DDRB`. The compiler stands no chance to optimize the readback away, as an IO port register is declared "volatile". Thus, chaining that kind of IO port assignments would better be avoided, using explicit assignments instead:

```
DDRB = 0xff;
DDRD = 0xff;
```

Even worse ist this, e. g. on an ATmega1281:

```
DDRA = DDRB = DDRC = DDRD = DDRE = DDRF = DDRG = 0xff;
```

The same happens as outlined above. However, when reading back register `DDRG`, this register only implements 6 out of the 8 bits, so the two topmost (unimplemented) bits read back as 0! Consequently, all remaining `DDR`*x* registers get assigned the value 0x3f, which does not match the intention of the developer in any way.

11.35 Which header files are included in my program?

Suppose we have a simple program like
```
#include <avr/pgmspace.h>

int main (void)
{
    return 0;
}
```

and we want to know which files this `#include` triggers. Just add option `-H` to the compiler options and check what is printed on standard output:

```
$ avr-gcc -H -S main.c -mmcu=atmega8
. <install>/avr/include/avr/pgmspace.h
.. <install>/avr/include/inttypes.h
... <install>/lib/gcc/avr/<version>/include/stdint.h
.... <install>/avr/include/stdint.h
.. <install>/lib/gcc/avr/<version>/include/stddef.h
.. <install>/avr/include/avr/io.h
... <install>/avr/include/avr/sfr_defs.h
... <install>/avr/include/avr/iom8.h
... <install>/avr/include/avr/portpins.h
...
```

where `<install>` denotes the installation path, `<version>` denotes the GCC version, and the number of dots indicates the include level, e.g. `inttypes.h` is included by `pgmspace.h`.

When `-v` is added to the compiler options, then the search paths are also displayed (amongst other stuff):

```
#include "..." search starts here:
#include <...> search starts here:
 <install>/bin/../lib/gcc/avr/<version>/include
 <install>/bin/../lib/gcc/avr/<version>/include-fixed
 <install>/bin/../lib/gcc/avr/<version>/../../../../avr/include
End of search list.
```

After resolving the `..`'s for *"parent directory"*, the last directory becomes
`<install>/avr/include`.

Back to FAQ Index.

11.36 Which macros are defined in my program? Where are they defined, and to what value?

One way is to add `-save-temps` and `-g3` to the compiler options. This saves the temporary files like the preprocessed source code in an `.i` file (for C sources), an `.ii` (for C++), or a `.s` (for assembly). A debug level of DWARF3 or higher is required to include the macro definitions in the file, with lower debug levels, only the preprocessed source will be present.

For a module with a simple `#include <avr/pgmspace.h>`, the saved intermediate file might look something like:

```
# 0 "<built-in>"
#define __STDC__ 1
```

The `__STDC__` macro is defined built-in in the compiler.

```
# 0 "<command-line>"
#define __AVR_DEVICE_NAME__ atmega8
```

The __AVR_DEVICE_NAME__ macro is defined on the command line by means of `-D __AVR_DEVICE_NAME__=atmega8`. In this special case, the `-D` option is added by the specs file `specs-atmega8`.

```
# 81 "<install>/avr/include/avr/pgmspace.h" 3
#define __PGMSPACE_H_ 1


#define __need_size_t
```

The __PGMSPACE_H_ macro is defined in line 81 of that header file. When there is no line note directly above the definition, go up until you find a line note. For example, the `__need_size_t` macro is defined in line 84 of that file.

Back to FAQ Index.

11.37 What ISR names are available for my device?

One way to find the possible ISR names is to pre-process a small file, and to grep for possible names, like in:

```
$ echo "#include <avr/io.h>" | avr-gcc -xc - -mmcu=atmega8 -E -dM  | grep _VECTOR
#define INT0_vect _VECTOR(1)
#define INT1_vect _VECTOR(2)
#define TIMER2_COMP_vect _VECTOR(3)
#define TIMER2_OVF_vect _VECTOR(4)
#define TIMER1_CAPT_vect _VECTOR(5)
...
```

Explanation:

`echo "#include <avr/io.h>"` This prints `#include <avr/io.h>` to the standard output, which is picked up by the following command as a C program to be preprocessed.

`avr-gcc -xc - -mmcu=atmega8 -E -dM` Set the input language to C, read the program from standard input (specified by a dash), preprocess, and print all macro definitions to the standard output.

`grep _VECTOR` Only print lines with `_VECTOR` in them.
The output above was actually generated with an additional `| sort -t '(' -k 2n` so that the vectors are printed in order.

In order to find the respective vector numbers, use `grep _vect_num` instead.

Back to FAQ Index.

12 Building and Installing the GNU Tool Chain

This chapter shows how to build and install, from source code, a complete development environment for the AVR processors using the GNU toolset. There are two main sections, one for Linux, FreeBSD, and other Unix-like operating systems, and another section for Windows.

12.1 Required AVR Tools

GNU Binutils Project Home: `https://sourceware.org/binutils`
Source Downloads: `https://sourceware.org/pub/binutils/releases`
FTP: anonymous@ftp.gnu.org/gnu/binutils
Git: git://sourceware.org/git/binutils-gdb.git
GitHub Mirror: `https://github.com/bminor/binutils-gdb`
Installation

GCC Project Home `https://gcc.gnu.org`
Mirrors Site: `https://gcc.gnu.org/mirrors.html`
FTP: anonymous@ftp.gnu.org/gnu/gcc
Git: git://gcc.gnu.org/git/gcc.git
GitHub Mirror: `https://github.com/gcc-mirror/gcc`
Installation: `https://gcc.gnu.org/install`
Installation

AVR-LibC Project Home: `http://savannah.gnu.org/projects/avr-libc`
Source Downloads: `https://download-mirror.savannah.gnu.org/releases/avr-libc`

Git: https://github.com/avrdudes/avr-libc.git
GitHub: `https://github.com/avrdudes/avr-libc`
Installation

12.2 Optional AVR Tools

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

AVRDUDE Project Home: http://savannah.nongnu.org/projects/avrdude
Git: https://github.com/avrdudes/avrdude.git
GitHub: https://github.com/avrdudes/avrdude
Installation
Usage Notes

GDB The GNU Debugger GDB is hosted together with GNU Binutils. When you don't want or need GDB, you can configure Binutils with `--disable-gdb`.

SimulAVR http://savannah.gnu.org/projects/simulavr
Installation

AVaRICE GitHub: https://github.com/avrdudes/avarice
Installation

12.3 Building and Installing under Linux, FreeBSD, and Others

The default behaviour for most of these tools is to install every thing under the `/usr/local` directory. In order to keep the AVR tools separate from the base system, it is usually better to install everything into `/usr/local/avr`. If the `/usr/local/avr` directory does not exist, you should create it before trying to install anything. You will need `root` access to install there. If you don't have or want root access to the system, you can alternatively install in your home directory, for example, in `$HOME/local/avr`. Where you install is a completely arbitrary decision, but should be consistent for all the tools.

Warning

If you have `CC` set to anything other than `avr-gcc` in your environment, this will cause the configure script to fail. It is best to not have `CC` set at all.

Note

It is usually the best to use the latest released version of each of the tools.

12.3.1 Preparations

12.3.1.1 Install Location You specify the installation directory by using the `--prefix=dir` option with the `configure` script. It is important to install all the AVR tools in the same directory or some of the tools will not work correctly. To ensure consistency and simplify the discussion, we will use `$PREFIX` to refer to whatever directory you wish to install in. You can set this as an environment variable if you wish as such (using a Bourne-like shell):

```
$ PREFIX=$HOME/local/avr
$ export PREFIX
```

Note

Be sure that you have your `PATH` environment variable set to search the directory you install everything in *before* you start installing anything. For example, if you use `--prefix=$PREFIX`, you must have `$PREFIX/bin` in your exported `PATH`. As such:

```
$ PATH=$PATH:$PREFIX/bin
$ export PATH
```

12.3.1.2 Directory Layout The instructions below build Binutils, GCC and AVR-LibC *outside* of the source tree, because:

- When something goes wrong, you can just remove the build directory and start all over again with a fresh build folder.
- You may want to build the tools with different configure options, e.g. build the tools for a Linux host and then build a Canadian cross to run on a Windows host.
- GCC does not support configuring anywhere in the source tree, so we'll have to use a separate build folder outside the source tree, anyway.

The instructions below assume that you have set up a directory tree like

```
+--source
+--build
```

in some place where you have write access, like in your home directory.

After successful downloads and builds, the tree will be something like:

```
+--source
|    +--gcc-<version>
|    +--binutils-<version>
|    +--avr-libc-<version>
+-- build
     +--gcc-<version>-avr
     +--binutils-<version>-avr
     +--avr-libc-<version>
```

12.3.2 GNU Binutils for the AVR target

The **Binutils** package provides all the low-level utilities needed in building and manipulating object files. Once installed, your environment will have an AVR assembler (`avr-as`), linker (`avr-ld`), and librarian (`avr-ar` and `avr-ranlib`). In addition, you get tools which extract data from object files (`avr-objcopy`), dissassemble object file information (`avr-objdump`), and strip information from object files (`avr-strip`). Before we can build the C compiler, these tools need to be in place.

Download and unpack the source files:

```
$ # in ./source
$ tar xfj binutils-<version>.tar.bz2
```

Replace `<version>` with the version of the package you downloaded.

If you obtained a gzip compressed file (`.tar.gz` or .tgz), use `gunzip` instead of `bunzip2`, or `tar xfz` *`file`*`.tar.gz`.

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. When you also want GDB, just drop `--disable-gdb`.

```
$ # in ./build
$ mkdir binutils-<version>-avr
$ cd binutils-<version>-avr
$ ../../source/binutils-<version>/configure --prefix=$PREFIX --target=avr \
    --disable-nls --disable-sim --disable-gdb --disable-werror
```

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 and that are run with the `make` command.

```
$ make
```

Note

BSD users should note that the project's Makefile uses GNU make syntax. This means FreeBSD users may need to build the tools by using gmake.

If the tools compiled cleanly, you're ready to install them. If you specified a destination that isn't owned by your account, you'll need root access to install them. To install:

```
$ make install
```

You should now have the programs from Binutils installed into $PREFIX/bin. Don't forget to set your PATH environment variable before going to build avr-gcc. To check that the correct assembler is found, run

```
$ avr-as --version
```

which should print the <version> of the used Binutils sources.

12.3.3 GCC for the AVR target

Warning

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

Before we can configure the compiler, we have to prepare the sources. GCC depends on some external host libraries, namely GMP, MPFR, MPC and ISL. You can build and install the appropriate versions of the required prerequisites by hand and provide their location by means of --with-gmp= etc. Though in most situations it is easier to let GCC download and build these libraries as part of the configure and build process. All what's needed is an internet connection when running ./contrib/download_prerequisites:

```
$ # in ./source
$ tar xfj gcc-<version>.tar.bz2
$ cd gcc-<version>
$ ./contrib/download_prerequisites

$ # in ./build
$ mkdir gcc-<version>-avr
$ cd gcc-<version>-avr
$ ../../source/gcc-<version>/configure --prefix=$PREFIX --target=avr --enable-languages=c,c++ \
    --disable-nls --disable-libssp --disable-libcc1 \
    --with-gnu-as --with-gnu-ld --with-dwarf2
$ make
$ make install # or make install-strip
```

The GCC binaries may consume quite some disc space. In most cases, you don't need the debug information in the compiler proper, and installing with

```
$ make install-strip
```

can save you some space.

12.3.4 AVR-LibC

Warning

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

Note

If you have obtained the latest AVR-LibC from git, you will have to run the `./bootstrap` script before using either of the build methods described below.

To build and install AVR-LibC:

```
$ # in ./source
$ tar xfz avr-libc-<version>.tar.gz

$ # in ./build
$ mkdir avr-libc-<version>
$ cd avr-libc-<version>
$ ../../source/avr-libc-<version>/configure --prefix=$PREFIX \
    --build=x86_64-pc-linux-gnu --host=avr
$ make
$ make install
```

Where the `--build` platform can be guessed by running

```
$ ./source/avr-libc-<version>/config.guess
```

12.3.5 AVRDUDE

Note

It has been ported to windows (via MinGW or cygwin), Linux and Solaris. Other Unix systems should be trivial to port to.

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

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

Note

Installation into the default location usually requires root permissions. However, running the program only requires access permissions to the appropriate `ppi(4)` device.

Building and installing on other systems should use the `configure` system, as such:

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

12.3.6 SimulAVR

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

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

Note

You might want to have already installed avr-binutils, avr-gcc and AVR-LibC if you want to have the test programs built in the simulavr source.

12.3.7 AVaRICE

Note

These install notes are not applicable to avarice-1.5 or older. You probably don't want to use anything that old anyways since there have been many improvements and bug fixes since the 1.5 release.

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

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

Note

AVaRICE uses the BFD library for accessing various binary file formats. You may need to tell the configure script where to find the lib and headers for the link to work. This is usually done by invoking the configure script like this (Replace <hdr_path> with the path to the `bfd.h` file on your system. Replace <lib_path> with the path to `libbfd.a` on your system.):

```
$ CPPFLAGS=-I<hdr_path> LDFLAGS=-L<lib_path> ../configure --prefix=$PREFIX
```

12.4 Building and Installing under Windows

Building and installing the toolchain under Windows requires more effort because all of the tools required for building, and the programs themselves, are mainly designed for running under a POSIX environment such as Unix and Linux. Windows does not natively provide such an environment.

There are two projects available that provide such an environment, Cygwin and MinGW. There are advantages and disadvantages to both. Cygwin provides a very complete POSIX environment that allows one to build many Linux based tools from source with very little or no source modifications. However, POSIX functionality is provided in the form of a DLL that is linked to the application. This DLL has to be redistributed with your application and there are issues if the Cygwin DLL already exists on the installation system and different versions of the DLL. On the other hand, MinGW can compile code as native Win32 applications. However, this means that programs designed for Unix and Linux (i.e. that use POSIX functionality) will not compile as MinGW does not provide that POSIX layer for you. Therefore most programs that compile on both types of host systems, usually must provide some sort of abstraction layer to allow an application to be built cross-platform.

MinGW does provide somewhat of a POSIX environment, called MSYS, that allows you to build Unix and Linux applications as they would normally do, with a `configure` step and a `make` step. Cygwin also provides such an environment. This means that building the AVR toolchain is very similar to how it is built in Linux, described above. The main differences are in what the PATH environment variable gets set to, pathname differences, and the tools that are required to build the projects under Windows. We'll take a look at the tools next.

12.4.1 Tools Required for Building the Toolchain for Windows

These are the tools that are currently used to build an AVR tool chain. This list may change, either the version of the tools, or the tools themselves, as improvements are made.

MinGW Download the MinGW Automated Installer, 2013-10-04 (or later) https://sourceforge.net/projects/mingw/files

- Run `mingw-get-setup.exe`
- In the installation wizard, keep the default values and press the "Next" button for all installer pages except for the pages explicitly listed below.
- In the installer page "Repository Catalogues", select the "Download latest repository catalogues" radio button, and press the "Next" button
- In the installer page "License Agreement", select the "I accept the agreement" radio button, and press the "Next" button
- In the installer page "Select Components", be sure to select these items:
  - C compiler (default checked)
  - C++ compiler
  - Ada compiler
  - MinGW Developer Toolkit (which includes "MSYS Basic System").
- Install.

Install Cygwin Install everything, all users, UNIX line endings. This will take a *long* time. A fat internet pipe is highly recommended. It is also recommended that you download all to a directory first, and then install from that directory to your machine.

Note

GMP, MPFR, MPC and ISL are required to build GCC. By far the easiest way to use them is by letting GCC download the sources locally by means of running the `./contrib/download_prewrequisites` script from the GCC top source. GCC will configure and build these libs during `configure` and `make`.

Doxygen is required to build AVR-LibC documentation.

- **Install Doxygen**
  - Version 1.7.2
  - https://www.doxygen.nl
  - Download and install.

NetPBM is required to build graphics in the AVR-LibC documentation.

- **Install NetPBM**
  - Version 10.27.0
  - From the GNUWin32 project: http://gnuwin32.sourceforge.net/packages.html
  - Download and install.

fig2dev is required to build graphics in the AVR-LibC documentation.

- **Install fig2dev**
  - Version 3.2 patchlevel 5c
  - From WinFig 4.62: http://winfig.com/downloads

  - Download the zip file version of WinFig
  - Unzip the download file and install fig2dev.exe in a location of your choice, somewhere in the PATH.
  - You may have to unzip and install related DLL files for fig2dev. In the version above, you have to install QtCore4.dll and QtGui4.dll.

MikTeX is required to build various documentation.

- **Install MiKTeX**

  - Version 2.9
  - https://miktex.org
  - Download and install.

Ghostscript is required to build various documentation.

- **Install Ghostscript**

  - Version 9.00
  - https://www.ghostscript.com
  - Download and install.
  - In the \bin subdirectory of the installaion, copy gswin32c.exe to gs.exe.
- Set the `TEMP` and `TMP` environment variables to **`c:\\temp`** or to the short filename version. This helps to avoid NTVDM errors during building.

12.4.2 Building the Toolchain for Windows

All directories in the PATH environment variable should be specified using their short filename (8.3) version. This will also help to avoid NTVDM errors during building. These short filenames can be specific to each machine.

Build the tools below in MinGW/MSYS.

- **Binutils**

  - Open source code package and patch as necessary.
  - Configure and build in a directory outside of the source code tree.
  - Set PATH, in order:
    * <MikTex executables>
    * <ghostscript executables>
    * /usr/local/bin
    * /usr/bin
    * /bin
    * /mingw/bin
    * c:/cygwin/bin
    * <install directory>/bin
  - Configure

```
CFLAGS=-D__USE_MINGW_ACCESS  \
../$archivedir/configure \
    --prefix=$installdir \
    --target=avr \
    --disable-nls \
    --enable-doc \
    --datadir=$installdir/doc/binutils \
    2>&1 | tee binutils-configure.log
```

  - Make

    ```
    make all html install install-html 2>&1 | tee binutils-make.log
    ```

  - Manually change documentation location.

- **GCC**

  - Open source code pacakge and patch as necessary.
  - Configure and build in a directory outside of the source code tree.
  - Set PATH, in order:
    - <MikTex executables>
    - <ghostscript executables>
    - /usr/local/bin
    - /usr/bin
    - /bin
    - /mingw/bin
    - c:/cygwin/bin
    - <install directory>/bin
  - Configure

    ```
    LDFLAGS='-L /usr/local/lib -R /usr/local/lib' \
    CFLAGS='-D__USE_MINGW_ACCESS'  \
    ../gcc-$version/configure \
        --prefix=$installdir \
        --target=$target \
        --enable-languages=c,c++ \
        --with-dwarf2 \
        --enable-doc \
        --with-docdir=$installdir/doc/$project \
        --disable-shared \
        --disable-libada \
        --disable-libssp \
        --disable-libcc1 \
        --disable-nls \
        2>&1 | tee $project-configure.log
    ```

  - Make

    ```
    make all html install 2>&1 | tee $package-make.log
    ```

- **AVR-LibC**

  - Open source code package.
  - Configure and build at the top of the source code tree.
  - Set PATH, in order:
    - /usr/local/bin
    - /mingw/bin
    - /bin
    - <MikTex executables>
    - <install directory>/bin
    - <Doxygen executables>
    - <NetPBM executables>
    - <fig2dev executable>
    - <Ghostscript executables>
    - c:/cygwin/bin
  - Configure

```
./configure \
    --host=avr \
    --prefix=$installdir \
    --enable-doc \
    --disable-versioned-doc \
    --enable-html-doc \
    --enable-pdf-doc \
    --enable-man-doc \
    --mandir=$installdir/man \
    --datadir=$installdir \
    2>&1 | tee $package-configure.log
```

  - Make

```
make all install 2>&1 | tee $package-make.log
```

  - Manually change location of man page documentation.
  - Move the examples to the top level of the install tree.
  - Convert line endings in examples to Windows line endings.
  - Convert line endings in header files to Windows line endings.

- **AVRDUDE**

  - Open source code package.
  - Configure and build at the top of the source code tree.
  - Set PATH, in order:
    * <MikTex executables>
    * /usr/local/bin
    * /usr/bin
    * /bin
    * /mingw/bin
    * c:/cygwin/bin
    * <install directory>/bin
  - Set location of LibUSB headers and libraries

```
export CPPFLAGS="-I../../libusb-win32-device-bin-$libusb_version/include"
export CFLAGS="-I../../libusb-win32-device-bin-$libusb_version/include"
export LDFLAGS="-L../../libusb-win32-device-bin-$libusb_version/lib/gcc"
```

  - Configure

```
./configure \
    --prefix=$installdir \
    --datadir=$installdir \
    --sysconfdir=$installdir/bin \
    --enable-doc \
    --disable-versioned-doc \
    2>&1 | tee $package-configure.log
```

  - Make

```
make -k all install 2>&1 | tee $package-make.log
```

  - Convert line endings in avrdude config file to Windows line endings.
  - Delete backup copy of avrdude config file in install directory if exists.

- **Insight/GDB**

  - Open source code pacakge and patch as necessary.
  - Configure and build in a directory outside of the source code tree.
  - Set PATH, in order:

        * <MikTex executables>
        * /usr/local/bin
        * /usr/bin
        * /bin
        * /mingw/bin
        * c:/cygwin/bin
        * <install directory>/bin
    - Configure

```
CFLAGS=-D__USE_MINGW_ACCESS  \
LDFLAGS='-static' \
../$archivedir/configure \
    --prefix=$installdir \
    --target=avr \
     --with-gmp=/usr/local \
     --with-mpfr=/usr/local \
     --enable-doc \
     2>&1 | tee insight-configure.log
```

    - Make

```
make all install 2>&1 | tee $package-make.log
```

- **SRecord**

    - Open source code package.
    - Configure and build at the top of the source code tree.
    - Set PATH, in order:
        * <MikTex executables>
        * /usr/local/bin
        * /usr/bin
        * /bin
        * /mingw/bin
        * c:/cygwin/bin
        * <install directory>/bin
    - Configure

```
./configure \
    --prefix=$installdir \
    --infodir=$installdir/info \
    --mandir=$installdir/man \
    2>&1 | tee $package-configure.log
```

    - Make

```
make all install 2>&1 | tee $package-make.log
```

Build the tools below in Cygwin.

- **AVaRICE**

    - Open source code package.
    - Configure and build in a directory outside of the source code tree.
    - Set PATH, in order:
        * <MikTex executables>
        * /usr/local/bin

    * /usr/bin
    * /bin
    * <install directory>/bin
  – Set location of LibUSB headers and libraries

```
export CPPFLAGS=-I$startdir/libusb-win32-device-bin-$libusb_version/include
export CFLAGS=-I$startdir/libusb-win32-device-bin-$libusb_version/include
export LDFLAGS="-static -L$startdir/libusb-win32-device-bin-$libusb_version/lib/gcc "
```

  – Configure

```
../$archivedir/configure \
    --prefix=$installdir \
    --datadir=$installdir/doc \
    --mandir=$installdir/man \
    --infodir=$installdir/info \
    2>&1 | tee avarice-configure.log
```

  – Make

```
make all install 2>&1 | tee avarice-make.log
```

- **SimulAVR**

  – Open source code package.
  – Configure and build in a directory outside of the source code tree.
  – Set PATH, in order:
    * <MikTex executables>
    * /usr/local/bin
    * /usr/bin
    * /bin
    * <install directory>/bin
  – Configure

```
export LDFLAGS="-static"
../$archivedir/configure \
    --prefix=$installdir \
    --datadir=$installdir \
    --disable-tests \
    --disable-versioned-doc \
    2>&1 | tee simulavr-configure.log
```

  – Make

```
make -k all install 2>&1 | tee simulavr-make.log
make pdf install-pdf 2>&1 | tee simulavr-pdf-make.log
```

12.5 Canadian Cross Builds

It is also possible to build avr-gcc for host Windows on a Linux build system. Suppose you have installed a `i686-w64-mingw32-gcc` toolchain that can compile code to run on `host=i686-w64-mingw32`. Then the steps to build a toolchain for Windows are:

1. Build and install the AVR toolchain for the Linux build machine as explained above. Make sure that running the command

```
avr-gcc --version
```

prints the compiler version according to the used GCC sources. The native AVR cross compiler is required during configure and to build the AVR target libraries like libgcc. Similarly, the version of the found AVR Binutils programs must match the version of the used Binutils sources.

2. Determine the name of the `--build` platform like `x86_64-pc-linux-gnu`, for example by running the `config.guess` script as shipped with the top level GCC sources (and also with Binutils sources, and AVR-LibC sources after `./bootstrap`).
3. Use different build and install directories, e.g. `./build/binutils-<version>-avr-mingw32` to build Binutils and `--prefix=$PREFIX-mingw32` as install path.
4. Configure, build and install **Binutils** and **GCC** like for the native build, but add the following configure options:

   ```
   --build=x86_64-pc-linux-gnu --host=i686-w64-mingw32
   ```

   This assumes that the required host libraries like GMP are being built in one go with the compiler. This is accomplished by running the `contrib/download_prerequisites` script from the toplevel GCC sources, just like with the native build.
5. There is no need to build **AVR-LibC** again because it is a pure target library. It can be installed by means of running

   ```
   $ # in ./build/avr-libc-<version>
   $ make install prefix=$PREFIX-mingw32
   ```

In order to "install" the toolchain on Windows, the canadian cross installed in `$PREFIX-mingw32` can be moved to the desired location on the Windows computer. The compiler can be used by calling it by its absolute path, or by adding the `$PREFIX-mingw32/bin` directory to the `PATH` environment variable.

12.6 Using Git

Most of the sources of the projects above are now managed with the `git` distributed version-control tools. When you want to build from the newest development branch, you can clone the repo, like with

```
$ git clone <repo> [dirname]
```

Replace `<repo>` with the URL of the Git repository, e.g. `https://github.com/avrdudes/avr-libc.git` for AVR-LibC. Notice that when building AVR-LibC from the repo source, you have to run `./bootstrap` from the top level AVR-LibC sources prior to `configure`.

Useful options for git clone:

`dirname` Specify an optional directory name for the cloned repository, like:

```
$ git clone https://github.com/avrdudes/avr-libc.git ./source/avr-libc-main
```

Without `dirname`, the name of the git file like `avr-libc` is used.

`--depth 1` An ordinary clone will clone the complete repository with all its branches and their history. To speed up the cloning and save some disc space, you can just clone the top of the history to some depth.

`--branch` ***`branch`*** The default branch is the head of the latest development, which is `master` for GCC and Binutils, and `main` for AVR-LibC.

When you want a different ref, like GCC's `releases/gcc-14` for the head of the GCC v14 branch, or `releases/gcc-14.1.0` for the GCC v14.1 release tag, then you can specify that as *`branch`*. To see the available refs, you can use

```
$ git ls-remote <repo>
```

13 Using the GNU tools

This is a short summary of the AVR-specific aspects of using the GNU tools. Normally, the generic documentation of these tools is fairly large and maintained in `texinfo` files. Command-line options are explained in detail in the manual page.

13.1 Options for the C compiler avr-gcc

13.1.1 Machine-specific options for the AVR

The following machine-specific options are recognized by the C compiler frontend. The preprocessor will define the macros `__AVR` and `__AVR__` (to the value 1) when compiling for an AVR target. The macro `AVR` will be defined as well, except in strict ANSI mode.

There are many options supported by avr-gcc, which also depend on the compiler version. For a complete overview, please see the documentation of avr-gcc's command line options. Here are links to supported options of the respective release series:

- `Current development` (work in progress)
- `v14`
- `v13.2`, `v13.3`
- `v12.3`, `v12.4`
- `v11`
- `v10`
- `v9`
- `v8`
- `v7`
- `v6`
- `v5`
- `v4.9`
- `v4.8`
- `v4.7`

Apart from the documentation of command line options, the linked pages also contain:

- The documentation of `built-in macros` like `__AVR_ARCH__`, `__AVR_ATmega328P__` and `__AVR_HAVE_MUL__`, just to mention a few.
- How the compiler treats the `RAMPX, RAMPY, RAMPZ and RAMPD special function registers` on devices that have (one of) them.
- How the compiler treats the `EIND special function register` on devices with more than 128 KiB of program memory, and how indirect calls are realized on such devices.

-mmcu=*arch*

-mmcu=*mcu* Compile code for architecture *arch* resp. AVR device *mcu*.

Since GCC v5, the compiler no more supports indivitual devices, instead the compiler comes with `device specs files` that describe which options to use with each sub-processes like pre-processor, compiler proper, assembler and linker.

The purpose of these specs files is to add support for AVR devices that the compiler does not yet support.

The easiest way to add support for an unsupported device is to use device support from an `atpack archive` as provided by the hardware manufacturer. Apart from the *mcu* specific specs file, it provides device headers `io*.h`, startup code `crt`*mcu*`.o` and device library `lib`*mcu*`.a`.

13.1.2 Selected general compiler options

The following general gcc options might be of some interest to AVR users.

-O*n* Optimization level *n*. Increasing *n* is meant to optimize more.

-O0 reduces compilation time and makes debugging produce the expected results. This is the default. Turning off all optimizations will prevent some warnings from being issued since the generation of those warnings depends on code analysis steps that are only performed when optimizing (unreachable code, unused variables). Moreover, the delay routines in `<util/delay.h>` require optimization to be turned on.

-O2 optimizes for speed, but without enabling very expensive optimizations like `-O3` does.

-Os turns on all `-O2` optimizations except those that often increase code size. In most cases, this is the preferred optimization level for AVR programs.

-Og optimizes debugging experience. This should be the optimization level of choice for the standard edit-compile-debug cycle, offering a reasonable level of optimization while maintaining fast compilation and a good debugging experience.

-O3 attempts to inline all "simple" functions and might unroll some loops. For the AVR target, this will normally constitute a large pessimization due to the code increasement.

-O is equivalent to **-O1**. The compiler tries to reduce code size and execution time, without performing any optimizations that take a great deal of compilation time.

See also the appropriate FAQ entry for issues regarding debugging optimized code.

-Wp,*preprocessor-options*

-Wa,*assembler-options*

-Wl,*linker-options* Pass the listed options to the C preprocessor, the assembler or the linker, respectively. Several options can be passed at once when they are separated by a , (comma).

-g Generate debugging information that can be used by avr-gdb. GCC v12 changed the default from STABS to DWARF. Different DWARF levels can be selected by `-g2` or `-gdwarf-3`.
The compiler may use GNU extensions to the DWARF format. When a debugger has problems with that, try `-gstrict-dwarf`.

-x *lang*

-x none Compile the following files in language *lang*. Values for *lang* are: `c`, `c++`, `assembler`, `assembler-with-cpp` and `none`.

For example, GCC does not recognize the `.asm` file extension as assembly source. With `-x assembler-with-cpp file.asm`, the compiler first runs the C preprocessor on `file.asm` (so that `#include <avr/io.h>` can be used in assembly), and then calls the assembler.

Another use case is to compile a C file that's read from standard input, which is specified as `-` (dash) instead of the name of a source file. As no source file name is specified, the compiler must be told which language to use: The command

```
$ echo '#include <avr/io.h>' | avr-gcc -xc - -mmcu=atmega8 -E -dM | grep _VECTOR
```

pre-processes the C file `#include <avr/io.h>` and writes all macro definitions to stdout. The output is then filtered by `grep` to show all possible ISR vector names for ATmega8.

`-x none` returns to the default for the following inputs, i.e. infer the respective source languages from the file extensions.

-save-temps

-save-temps=obj

-save-temps=cwd

-fverbose-asm

-dumpbase *base*

-dumpdir *dir* Don't remove temporary, intermediate files like C preprocessor output and assembly code generated by the compiler. The intermediate files have file extension `.i` (preprocessed C), `.ii` (preprocessed C++) and `.s` (preprocessed assembly, compiler-generated assembly).

`-dumpbase` and `-dumpdir` can be used to adjust file names and locations.

With `-fverbose-asm`, the compiler adds the high level source code to the assembly output. Compiling without debugging information (`-g0`) improves legibility of the generated assembly.

The preprocessed files can be used to check if macro expansions work as expected. With `-g3` or higher, the preprocessed files will also contain all macro definitions and indications where they are defined: Built-in, on the command line, or in some header file as indicated by `#line` notes.

-l*name* Locate the archive library named `lib`*name*`.a`, and use it to resolve currently unresolved symbols from it. The library is searched along a path that consists of builtin pathname entries that have been specified at configure time (e. g. `/usr/local/avr/lib` on Unix systems), possibly extended by pathname entries as specified by `-L` options (that must precede the `-l` options on the command-line).

-L*path* Additional directory *path* to look for archive libraries requested by `-l` options.

-ffunction-sections

-fdata-sections Put each function resp. object in static storage in its own input section. This is used with `-Wl,--gc-sections` so the linker can better garbage-collect unused sections, which are sections that are neither referenced by other sections, nor are marked as `KEEP`, nor are referenced by an entry symbol.

-mrelax Replace `JMP` and `CALL` instructions with the faster and shorter `RJMP` and `RCALL` instructions if possible. That optimization is performed by the linker, and the assembler must not resolve some expressions, which is all arranged by `-mrelax`.

-Tbss *org*

-Tdata *org*

-Ttext *org* Start the `.bss`, `.data`, or `.text` section at VMA address *org*, respectively.

-T *scriptfile* Use *scriptfile* as the linker script, replacing or augmenting the default linker script.

Default linker scripts are stored in a system-specific location (e. g. under `/usr/local/avr/lib/ldscripts` on Unix systems), and consist of the AVR architecture name (`avr2` through `avrxmega7`) with the suffix `.x` appended. They describe how the various memory sections will be linked together and which input sections go into them. Notice that the default linker scripts are part of the linker binary, changing them on file will have no effect.

For a simple linker script augmentation, see the avr-gcc Wiki on `named address spaces`.

-nostdlib Don't link against standard libraries.

-nodefaultlibs Don't link against default libraries.

-nodevicelib Don't link against AVR-LibC's `lib`*mcu*`.a` that contains EEPROM support and other stuff. This can be used when no such library is available.

-nostartfiles Don't link against AVR-LibC's startup code crt*mcu*.o.

Notice that parts of the startup code are provided by libgcc.a. To get rid of that, one can -nostdlib or -nodefaultlibs; however that also removes other code like functions required for arithmetic. To just get rid of the startup bits, define the respective symbols, for example -Wl,--defsym,__do_clear_bss=0 and similar for __do_copy_data, __do_global_ctors and __do_global_dtors.

-funsigned-char This option changes the binary interface!
Make any unqualfied char type an unsigned char. Without this option, they default to a signed char.

-funsigned-bitfields This option changes the binary interface!
Make any unqualified bitfield type unsigned. By default, they are signed.

-fshort-enums This option changes the binary interface!
Allocate to an enum type only as many bytes as it needs for the declared range of possible values. Specifically, the enum type will be equivalent to the smallest integer type which has enough room.

-fpack-struct This option changes the binary interface!
Pack all structure members together without holes.

-fno-jump-tables Do not generate tablejump instructions. By default, jump tables can be used to optimize switch statements. When turned off, sequences of compare statements are used instead. Jump tables are usually faster to execute on average, but in particular for switch statements, where most of the jumps would go to the default label, they might waste a bit of flash memory.

Note: Sinve GCC v4.9.2, tablejump code uses the ELPM instruction to read from jump tables. In older version, use the -fno-jump-tables switch when compiling a bootloader for devices with more than 64 KiB of code memory.

-ffreestanding Assume a "freestanding" environment as per the C standard. This turns off automatic builtin functions (though they can still be reached by prepending __builtin_ to the actual function name). It also makes the compiler not complain when main() is declared with a void return type. (In most cases, main() won't even return anyway, and hence using a return type of int has no downsides at all).

However, this option also turns off all optimizations normally done by the compiler which assume that functions known by a certain name behave as described by the standard. For example, applying the function strlen() to a literal string will normally cause the compiler to immediately replace that call by the actual length of the string, while with -ffreestanding, it will always call strlen() at run-time.

13.2 Options for the assembler avr-as

Note

The preferred way to assemble a file is by means of using avr-gcc:

- avr-gcc, which is a driver program to call sub-programs like the compiler proper or the assembler, knows which options it has to add to the assembler's command line, like: -mmcu=*arch*, -mno-skip-bug, etc.
- avr-gcc will call the C preprocessor on the assembler input for sources with extensions .S and .sx. For other extensions, use

```
$ avr-gcc -x assembler-with-cpp file.asm ...
```

This allows to use C preprocessor directives like

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

in assembly sources.

13.2.1 Machine-specific assembler options

-mmcu=*architecture*

-mmcu=*mcu* `avr-as` does not support all *mcu*s supported by the compiler. As explained in the note above, the preferred way to run the assembler is by using the compiler driver `avr-gcc`.

-mall-opcodes Turns off opcode checking, and allows any possible AVR opcode to be assembled.

-mno-skip-bug Don't emit a warning when trying to skip a 2-word instruction with a `CPSE/SBIC/SBIS/`↩ `SBRC/SBRS` instruction. Early AVR devices suffered from a hardware bug where these instructions could not be properly skipped.

-mno-wrap For `RJMP/RCALL` instructions, don't allow the target address to wrap around for devices that have more than 8 KiB of memory.

--gstabs Generate `.stabs` debugging symbols for assembler source lines. This enables `avr-gdb` to trace through assembler source files. This option *must not* be used when assembling sources that have been generated by the C compiler; these files already contain the appropriate line number information from the C source files.

-a[cdhlmns=*file*] Turn on the assembler listing. The sub-options are:

- `c` omit false conditionals
- `d` omit debugging directives
- `h` include high-level source
- `l` include assembly
- `m` include macro expansions
- `n` omit forms processing
- `s` include symbols
- `=`*`file`* set the name of the listing file

The various sub-options can be combined into a single `-a` option list; *=file* must be the last one in that case.

13.2.2 Examples for assembler options passed through the C compiler

Remember that assembler options can be passed from the C compiler frontend using `-Wa` (see gcc_minusW above), so in order to include the C source code into the assembler listing in file `foo.lss`, when compiling `foo.c`, the following compiler command-line can be used:

```
$ avr-gcc -mmcu=atmega8 -c foo.c -o foo.o -Wa,-ahls=foo.lss
```

In order to pass an assembler file through the C preprocessor first, and have the assembler generate line number debugging information for it, the following command can be used:

```
$ avr-gcc -c -x assembler-with-cpp -o foo.o foo.asm -Wa,--gstabs
```

Note that on Unix systems that have case-distinguishing file systems, specifying a file name with the suffix `.S` (upper-case letter `S`) will make the compiler automatically assume `-x assembler-with-cpp`, while using `.s` would pass the file directly to the assembler (no preprocessing done).

13.3 Controlling the linker avr-ld

Note

It is highly recommended to use the compiler driver `avr-gcc` or `avr-g++` for linking.

- The driver knows which options to pass down to the linker. This includes the correct multilib path, support libraries like `libc.a`, `libm.a`, `libgcc.a` and `lib`*mcu*`.a`, as well as options for LTO (link-time optimization) and options for plugins (that call back the compiler to compile LTO byte code), startup code and many more.
- The driver program understands options like `-l`*lib*, `-L`*path*, `Ttext`, `Tdata`, `Tbss` and `-T` *script*, so no `-Wl` is required for them.

13.3.1 Selected linker options

A number of the standard options might be of interest to AVR users.

--defsym *symbol=expr* Define a global symbol *symbol* using *expr* as the value.

-M Print a linker map to `stdout`.

-Map *mapfile* Print a linker map to *mapfile*.

--cref Output a cross reference table to the map file (in case `-Map` is also present), or to `stdout`.

--gc-sections Only keep input sections that are referenced (by other sections or the entry symbol), and that are not marked as `KEEP` in the linker script. This is used to reduce code size, usually together with compiler option `-ffunction-sections` so that input section granularity is on function level rather than on the level of compilation units.

--section-start *sectionname=org* Start section *sectionname* at absolute address *org*.

--relax Don't use this option directly or per `-Wl,--relax`. Instead, link with `avr-gcc ... -mrelax`.

13.3.2 Passing linker options from the C compiler

By default, all unknown non-option arguments on the `avr-gcc` command-line (i. e., all filename arguments that don't have a suffix that is handled by `avr-gcc`) are passed straight to the linker. Thus, all files ending in `.o` (object files) and `.a` (object libraries) are provided to the linker.

System libraries are usually not passed by their explicit filename but rather using the `-l` option which uses an abbreviated form of the archive filename (see above). AVR-LibC ships system libraries, `libc.a`, `libm.a` and `lib`*mcu*`.a`. While the standard library `libc.a` will always be searched for unresolved references when the linker is started using the C compiler frontend (i. e., there's always at least one implied `-lc` option), the mathematics library `libm.a` is only automatically added in GCC v4.7 and above. On older versions, it needs to be explicitly requested using `-lm`.

Conventionally, Makefiles use the `make` macro `LDLIBS` to keep track of `-l` (and possibly `-L`) options that should only be appended to the C compiler command-line when linking the final binary. In contrast, the macro `LDFLAGS` is used to store other command-line options to the C compiler that should be passed as options during the linking stage. The difference is that options are placed early on the command-line, while libraries are put at the end since they are to be used to resolve global symbols that are still unresolved at this point.

Specific linker flags can be passed from the C compiler command-line using the `-Wl` compiler option, see gcc_minusW above. This option requires that there be no spaces in the appended linker option, while some of the linker options above (like `-Map` or `--defsym`) would require a space. In these situations, the space can be replaced by an equal sign as well. For example, the following command-line can be used to compile `foo.c` into an executable, and also produce a link map that contains a cross-reference list in the file `foo.map`:

```
$ avr-gcc -mmcu=atmega8 foo.c -o foo.elf -Wl,-Map,foo.map -Wl,--cref
```

Alternatively, a comma as a placeholder will be replaced by a space before passing the option to the linker. So for a device with external SRAM, the following command-line would cause the linker to place the data segment at address 0x2000 in the SRAM:

```
$ avr-gcc -mmcu=atmega128 foo.c -o foo.elf -Wl,-Tdata,0x802000
```

See the explanation of the data section for why 0x800000 needs to be added to the actual value. Note that the stack will still remain in internal RAM, through the symbol `__stack` that is provided by the run-time startup code. This is probably a good idea anyway (since internal RAM access is faster), and even required for some early devices that had hardware bugs preventing them from using a stack in external RAM. Note also that the heap for `malloc()` will still be placed after all the variables in the data section, so in this situation, no stack/heap collision can occur.

In order to relocate the stack from its default location at the top of interns RAM, the value of the symbol `__stack` can be changed on the linker command-line. As the linker is typically called from the compiler frontend, this can be achieved using a compiler option like
`-Wl,--defsym=__stack=0x8003ff`

The above will make the code use stack space from RAM address 0x3ff downwards. The amount of stack space available then depends on the bottom address of internal RAM for a particular device. It is the responsibility of the application to ensure the stack does not grow out of bounds, as well as to arrange for the stack to not collide with variable allocations made by the compiler (sections .data and .bss).

14 Compiler optimization

14.1 Problems with reordering code

Author

Jan Waclawek

Programs contain sequences of statements, and a naive compiler would execute them exactly in the order as they are written. But an optimizing compiler is free to *reorder* the statements — or even parts of them — if the resulting "net effect" is the same. The "measure" of the "net effect" is what the standard calls "side effects", and is accomplished exclusively through accesses (reads and writes) to variables qualified as `volatile`. So, as long as all volatile reads and writes are to the same addresses and in the same order (and writes write the same values), the program is correct, regardless of other operations in it. One important point to note here is, that time duration between consecutive volatile accesses is not considered at all.

Unfortunately, there are also operations which are not covered by volatile accesses. An example of this in AVR-GCC/AVR-LibC are the `cli()` and `sei()` macros defined in <avr/interrupt.h>, which convert directly to the respective assembler mnemonics through the `__asm__()` statement. They constitute a variable access by means of their memory clobber, and they are (implicitly) volatile because they don't have an output operand. So the compiler may not reorder these inline asm statements with respect to other memory accesses or volatile actions. However, such asm statementy may still be reordered with other statement that are neither volatile nor access memory.

Note that even a volatile asm instruction can be moved relative to other code, including across (expensive) arithmetic and jump instructions [...]

See also

http://gcc.gnu.org/onlinedocs/gcc/Extended-Asm.html

However, not even a volatile memory barrier like

```
__asm __volatile__ ("" ::: "memory");
```

keeps GCC from reordering non-volatile, non-memory accesses across such barriers. Peter Dannegger provided a nice example of this effect:

```
#define cli() __asm volatile( "cli" ::: "memory" )
#define sei() __asm volatile( "sei" ::: "memory" )

unsigned int ivar;

void test2 (unsigned int val)
{
  val = 65535U / val;

  cli();

  ivar = val;

  sei();
}
```

avr-gcc v5.4 or v14 compile with optimisations switched on (`-Os`) to

```
00000112 <test2>:
 112: bc 01       movw r22, r24
 114: f8 94       cli
 116: 8f ef       ldi r24, 0xFF ; 255
 118: 9f ef       ldi r25, 0xFF ; 255
 11a: 0e 94 96 00 call 0x12c ; 0x12c <__udivmodhi4>
 11e: 70 93 01 02 sts 0x0201, r23
 122: 60 93 00 02 sts 0x0200, r22
 126: 78 94       sei
 128: 08 95       ret
```

where the potentially slow division is moved across `cli()`, resulting in interrupts to be disabled longer than intended. Note, that the volatile access occurs in order with respect to `cli()` or `sei()`; so the "net effect" required by the standard is achieved as intended, it is "only" the timing which is off. However, for most of embedded applications, timing is an important, sometimes critical factor.

See also

https://www.mikrocontroller.net/topic/65923

Unfortunately, at the moment, in avr-gcc (nor in the C standard), there is no mechanism to enforce complete match of written and executed code ordering — except maybe of switching the optimization completely off (`-O0`), or writing all the critical code in assembly.

Note

The artifact with the `__udivmodhi4` function is specific to avr-gcc and how the compiler represents the division internally. On other target platforms that are using a library function for division or whatever expensive operation, this eccect will not occur. The reason is that avr-gcc does not represent the library call as a function call but rather like an ordinary instruction. Outcome is that the GCC middle-end concludes that the division is cheap (because the backend has an instruction for it) but in fact it's not.

A work around for the code from above would be to enforce that the division havvens prior to the `cli()`:

```
val = 65535U / val;
__asm __volatile__ ("" : "+r" (val));
cli();
```

- The `volatile` forces the asm statememt prior to the `cli`.
- The asm has `val` as input operand, hence the division must be carried out prior to the asm because `val` is set by the division.

Notice that this work around does not work in general due to a variety of reasons:

- The division might be located in an inlined function.
- The variable might be read-only or may not be appropriate as an asm operand.
- There may be more such instruction prior to the division, and it is not practical to treat all of them like this.

To sum it up:

- volatile memory barriers don't ensure statements with no volatile accesses to be reordered across the barrier

15 Using the avrdude program

Note

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

The avrdude program was previously called avrprog. The name was changed to avoid confusion with the avrprog program that Atmel ships with AvrStudio.

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

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

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

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

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

avrdude: AVR device initialized and ready to accept instructions

avrdude: Device signature = 0x1e9101

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

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

avrdude done.  Thank you.
```

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

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

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

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

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

avrdude> ?
>>> ?
Valid commands:

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

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

avrdude>
```

16 Acknowledgments

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

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

- Eric Weddington [eweddington@cso.atmel.com] for maintaining the **WinAVR** package and thus making the continued improvements to the open source AVR toolchain available to many users.
- Rich Neswold for writing the original avr-tools document (which he graciously allowed to be merged into this document) and his improvements to the demo project.
- Theodore A. Roth for having been a long-time maintainer of many of the tools (**AVR-LibC**, the AVR port of **GDB**, **AVaRICE**, **uisp**, **avrdude**).
- All the people who currently maintain the tools, and/or have submitted suggestions, patches and bug reports. (See the AUTHORS files of the various tools.)
- And lastly, all the users who use the software. If nobody used the software, we would probably not be very motivated to continue to develop it. Keep those bug reports coming. ;-)

17 Deprecated List

Global cbi (port, bit)

Global enable_external_int (mask)

Global inb (port)

Global inp (port)

Global INTERRUPT (signame)

Global ISR_ALIAS (vector, target_vector)

For new code, the use of ISR(..., ISR_ALIASOF(...)) is recommended.

Global outb (port, val)

Global outp (val, port)

Global sbi (port, bit)

Global SIGNAL (vector)

Do not use SIGNAL() in new code. Use ISR() instead.

Global timer_enable_int (unsigned char ints)

18 Module Index

18.1 Modules

Here is a list of all modules:

19 Data Structure Index

19.1 Data Structures

Here are the data structures with brief descriptions:

20 File Index

20.1 File List

Here is a list of all documented files with brief descriptions:

21 Module Documentation

21.1 <alloca.h>: Allocate space in the stack

Functions

- void ∗ alloca (size_t __size)

21.1.1 Detailed Description

21.1.2 Function Documentation

21.1.2.1 alloca() `void * alloca ( size_t __size )`

Allocate *__size* bytes of space in the stack frame of the caller.

This temporary space is automatically freed when the function that called alloca() returns to its caller. AVR-LibC defines the alloca() as a macro, which is translated into the inlined `__builtin_alloca()` function. The fact that the code is inlined, means that it is impossible to take the address of this function, or to change its behaviour by linking with a different library.

Returns

alloca() returns a pointer to the beginning of the allocated space. If the allocation causes stack overflow, program behaviour is undefined.

Warning

Avoid use alloca() inside the list of arguments of a function call.

21.2 <assert.h>: Diagnostics

Macros

- #define assert(expression)

21.2.1 Detailed Description

```
#include <assert.h>
```

This header file defines a debugging aid.

As there is no standard error output stream available for many applications using this library, the generation of a printable error message is not enabled by default. These messages will only be generated if the application defines the macro

```
__ASSERT_USE_STDERR
```

before including the `<assert.h>` header file. By default, only abort() will be called to halt the application.

21.2.2 Macro Definition Documentation

21.2.2.1 assert `#define assert( expression )`

Parameters

expression	Expression to test for.

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

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

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

21.3 <ctype.h>: Character Operations

Character classification routines

These functions perform character classification. They return true or false status depending whether the character passed to the function falls into the function's classification (i.e. isdigit() returns true if its argument is any value '0' though '9', inclusive). If the input is not an unsigned char value, all of this function return false.

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

Character convertion routines

This realization permits all possible values of integer argument. The toascii() function clears all highest bits. The tolower() and toupper() functions return an input argument as is, if it is not an unsigned char value.

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

21.3.1 Detailed Description

These functions perform various operations on characters.

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

21.3.2 Function Documentation

21.3.2.1 isalnum() `int isalnum ( int __c )`

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

21.3.2.2 isalpha() `int isalpha ( int __c )`

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

21.3.2.3 isascii() `int isascii ( int __c )`

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

21.3.2.4 isblank() `int isblank ( int __c )`

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

21.3.2.5 iscntrl() `int iscntrl ( int __c )`

Checks for a control character.

21.3.2.6 isdigit() `int isdigit ( int __c )`

Checks for a digit (0 through 9).

21.3.2.7 isgraph() `int isgraph ( int __c )`

Checks for any printable character except space.

21.3.2.8 islower() `int islower ( int __c )`

Checks for a lower-case character.

21.3.2.9 isprint() `int isprint ( int __c )`

Checks for any printable character including space.

21.3.2.10 ispunct() `int ispunct ( int __c )`

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

21.3.2.11 isspace()

```
int isspace (
    int __c )
```

Checks for white-space characters. For the AVR-LibC library, these are: space, form-feed ('\f'), newline ('\n'), carriage return ('\r'), horizontal tab ('\t'), and vertical tab ('\v').

21.3.2.12 isupper()

```
int isupper (
    int __c )
```

Checks for an uppercase letter.

21.3.2.13 isxdigit()

```
int isxdigit (
    int __c )
```

Checks for a hexadecimal digits, i.e. one of 0 1 2 3 4 5 6 7 8 9 a b c d e f A B C D E F.

21.3.2.14 toascii()

```
int toascii (
    int __c )
```

Converts `c` to a 7-bit unsigned char value that fits into the ASCII character set, by clearing the high-order bits.

Warning

Many people will be unhappy if you use this function. This function will convert accented letters into random characters.

21.3.2.15 tolower()

```
int tolower (
    int __c )
```

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

21.3.2.16 toupper()

```
int toupper (
    int __c )
```

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

21.4 <errno.h>: System Errors

Macros

- #define EDOM 33
- #define ERANGE 34

Variables

- int errno

21.4.1 Detailed Description

```
#include <errno.h>
```

Some functions in the library set the global variable `errno` when an error occurs. The file, <`errno.h`>, provides symbolic names for various error codes.

21.4.2 Macro Definition Documentation

21.4.2.1 EDOM `#define EDOM 33`

Domain error.

21.4.2.2 ERANGE `#define ERANGE 34`

Range error.

21.4.3 Variable Documentation

21.4.3.1 errno `int errno [extern]`

Error code for last error encountered by library.

The variable `errno` holds the last error code encountered by a library function. This variable must be cleared by the user prior to calling a library function.

Warning

The `errno` global variable is not safe to use in a threaded or multi-task system. A race condition can occur if a task is interrupted between the call which sets `error` and when the task examines `errno`. If another task changes `errno` during this time, the result will be incorrect for the interrupted task.

21.5 <inttypes.h>: Integer Type conversions

Far pointers for memory access > 64K

- typedef int32_t int_farptr_t
- typedef uint32_t uint_farptr_t

macros for printf and scanf format specifiers

For C++, these are only included if __STDC_LIMIT_MACROS is defined before including <inttypes.h>.

- #define PRId8 "d"
- #define PRIdLEAST8 "d"
- #define PRIdFAST8 "d"
- #define PRIi8 "i"
- #define PRIiLEAST8 "i"
- #define PRIiFAST8 "i"
- #define PRId16 "d"
- #define PRIdLEAST16 "d"
- #define PRIdFAST16 "d"
- #define PRIi16 "i"
- #define PRIiLEAST16 "i"
- #define PRIiFAST16 "i"
- #define PRId32 "ld"
- #define PRIdLEAST32 "ld"
- #define PRIdFAST32 "ld"
- #define PRIi32 "li"
- #define PRIiLEAST32 "li"
- #define PRIiFAST32 "li"
- #define PRIdPTR PRId16
- #define PRIiPTR PRIi16
- #define PRIo8 "o"
- #define PRIoLEAST8 "o"
- #define PRIoFAST8 "o"
- #define PRIu8 "u"
- #define PRIuLEAST8 "u"
- #define PRIuFAST8 "u"
- #define PRIx8 "x"
- #define PRIxLEAST8 "x"
- #define PRIxFAST8 "x"
- #define PRIX8 "X"
- #define PRIXLEAST8 "X"
- #define PRIXFAST8 "X"
- #define PRIo16 "o"
- #define PRIoLEAST16 "o"
- #define PRIoFAST16 "o"
- #define PRIu16 "u"
- #define PRIuLEAST16 "u"
- #define PRIuFAST16 "u"
- #define PRIx16 "x"
- #define PRIxLEAST16 "x"
- #define PRIxFAST16 "x"
- #define PRIX16 "X"
- #define PRIXLEAST16 "X"
- #define PRIXFAST16 "X"
- #define PRIo32 "lo"
- #define PRIoLEAST32 "lo"
- #define PRIoFAST32 "lo"
- #define PRIu32 "lu"
- #define PRIuLEAST32 "lu"
- #define PRIuFAST32 "lu"

- #define PRIx32 "lx"
- #define PRIxLEAST32 "lx"
- #define PRIxFAST32 "lx"
- #define PRIX32 "lX"
- #define PRIXLEAST32 "lX"
- #define PRIXFAST32 "lX"
- #define PRIoPTR PRIo16
- #define PRIuPTR PRIu16
- #define PRIxPTR PRIx16
- #define PRIXPTR PRIX16
- #define SCNd8 "hhd"
- #define SCNdLEAST8 "hhd"
- #define SCNdFAST8 "hhd"
- #define SCNi8 "hhi"
- #define SCNiLEAST8 "hhi"
- #define SCNiFAST8 "hhi"
- #define SCNd16 "d"
- #define SCNdLEAST16 "d"
- #define SCNdFAST16 "d"
- #define SCNi16 "i"
- #define SCNiLEAST16 "i"
- #define SCNiFAST16 "i"
- #define SCNd32 "ld"
- #define SCNdLEAST32 "ld"
- #define SCNdFAST32 "ld"
- #define SCNi32 "li"
- #define SCNiLEAST32 "li"
- #define SCNiFAST32 "li"
- #define SCNdPTR SCNd16
- #define SCNiPTR SCNi16
- #define SCNo8 "hho"
- #define SCNoLEAST8 "hho"
- #define SCNoFAST8 "hho"
- #define SCNu8 "hhu"
- #define SCNuLEAST8 "hhu"
- #define SCNuFAST8 "hhu"
- #define SCNx8 "hhx"
- #define SCNxLEAST8 "hhx"
- #define SCNxFAST8 "hhx"
- #define SCNo16 "o"
- #define SCNoLEAST16 "o"
- #define SCNoFAST16 "o"
- #define SCNu16 "u"
- #define SCNuLEAST16 "u"
- #define SCNuFAST16 "u"
- #define SCNx16 "x"
- #define SCNxLEAST16 "x"
- #define SCNxFAST16 "x"
- #define SCNo32 "lo"
- #define SCNoLEAST32 "lo"
- #define SCNoFAST32 "lo"
- #define SCNu32 "lu"
- #define SCNuLEAST32 "lu"
- #define SCNuFAST32 "lu"
- #define SCNx32 "lx"

- #define SCNxLEAST32 "lx"
- #define SCNxFAST32 "lx"
- #define SCNoPTR SCNo16
- #define SCNuPTR SCNu16
- #define SCNxPTR SCNx16

21.5.1 Detailed Description

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

This header file includes the exact-width integer definitions from <stdint.h>, and extends them with additional facilities provided by the implementation.

Currently, the extensions include two additional integer types that could hold a "far" pointer (i.e. a code pointer that can address more than 64 KB), as well as standard names for all printf and scanf formatting options that are supported by the <stdio.h>: Standard IO facilities. As the library does not support the full range of conversion specifiers from ISO 9899:1999, only those conversions that are actually implemented will be listed here.

The idea behind these conversion macros is that, for each of the types defined by <stdint.h>, a macro will be supplied that portably allows formatting an object of that type in printf() or scanf() operations. Example:

```
#include <inttypes.h>

uint8_t smallval;
int32_t longval;
...
printf("The hexadecimal value of smallval is %" PRIx8
       ", the decimal value of longval is %" PRId32 ".\n",
   smallval, longval);
```

21.5.2 Macro Definition Documentation

21.5.2.1 PRId16 `#define PRId16 "d"`

decimal printf format for int16_t

21.5.2.2 PRId32 `#define PRId32 "ld"`

decimal printf format for int32_t

21.5.2.3 PRId8 `#define PRId8 "d"`

decimal printf format for int8_t

21.5.2.4 PRIdFAST16 `#define PRIdFAST16 "d"`

decimal printf format for int_fast16_t

21.5.2.5 PRIdFAST32 `#define PRIdFAST32 "ld"`

decimal printf format for int_fast32_t

21.5.2.6 PRIdFAST8 `#define PRIdFAST8 "d"`

decimal printf format for int_fast8_t

21.5.2.7 PRIdLEAST16 `#define PRIdLEAST16 "d"`

decimal printf format for int_least16_t

21.5.2.8 PRIdLEAST32 `#define PRIdLEAST32 "ld"`

decimal printf format for int_least32_t

21.5.2.9 PRIdLEAST8 `#define PRIdLEAST8 "d"`

decimal printf format for int_least8_t

21.5.2.10 PRIdPTR `#define PRIdPTR PRId16`

decimal printf format for intptr_t

21.5.2.11 PRIi16 `#define PRIi16 "i"`

integer printf format for int16_t

21.5.2.12 PRIi32 `#define PRIi32 "li"`

integer printf format for int32_t

21.5.2.13 PRIi8 `#define PRIi8 "i"`

integer printf format for int8_t

21.5.2.14 PRIiFAST16 `#define PRIiFAST16 "i"`

integer printf format for int_fast16_t

21.5.2.15 PRIiFAST32 `#define PRIiFAST32 "li"`

integer printf format for int_fast32_t

21.5.2.16 PRIiFAST8 `#define PRIiFAST8 "i"`

integer printf format for int_fast8_t

21.5.2.17 PRIiLEAST16 `#define PRIiLEAST16 "i"`

integer printf format for int_least16_t

21.5.2.18 PRIiLEAST32 `#define PRIiLEAST32 "li"`

integer printf format for int_least32_t

21.5.2.19 PRIiLEAST8 `#define PRIiLEAST8 "i"`

integer printf format for int_least8_t

21.5.2.20 PRIiPTR `#define PRIiPTR PRIi16`

integer printf format for intptr_t

21.5.2.21 PRIo16 `#define PRIo16 "o"`

octal printf format for uint16_t

21.5.2.22 PRIo32 `#define PRIo32 "lo"`

octal printf format for uint32_t

21.5.2.23 PRIo8 `#define PRIo8 "o"`

octal printf format for uint8_t

21.5.2.24 PRIoFAST16 `#define PRIoFAST16 "o"`

octal printf format for uint_fast16_t

21.5.2.25 PRIoFAST32 `#define PRIoFAST32 "lo"`

octal printf format for uint_fast32_t

21.5.2.26 PRIoFAST8 `#define PRIoFAST8 "o"`

octal printf format for uint_fast8_t

21.5.2.27 PRIoLEAST16 `#define PRIoLEAST16 "o"`

octal printf format for uint_least16_t

21.5.2.28 PRIoLEAST32 `#define PRIoLEAST32 "lo"`

octal printf format for uint_least32_t

21.5.2.29 PRIoLEAST8 `#define PRIoLEAST8 "o"`

octal printf format for uint_least8_t

21.5.2.30 PRIoPTR `#define PRIoPTR PRIo16`

octal printf format for uintptr_t

21.5.2.31 PRIu16 `#define PRIu16 "u"`

decimal printf format for uint16_t

21.5.2.32 PRIu32 `#define PRIu32 "lu"`

decimal printf format for uint32_t

21.5.2.33 PRIu8 `#define PRIu8 "u"`

decimal printf format for uint8_t

21.5.2.34 PRIuFAST16 `#define PRIuFAST16 "u"`

decimal printf format for uint_fast16_t

21.5.2.35 PRIuFAST32 `#define PRIuFAST32 "lu"`

decimal printf format for uint_fast32_t

21.5.2.36 PRIuFAST8 `#define PRIuFAST8 "u"`

decimal printf format for uint_fast8_t

21.5.2.37 PRIuLEAST16 `#define PRIuLEAST16 "u"`

decimal printf format for uint_least16_t

21.5.2.38 PRIuLEAST32 `#define PRIuLEAST32 "lu"`

decimal printf format for uint_least32_t

21.5.2.39 PRIuLEAST8 `#define PRIuLEAST8 "u"`

decimal printf format for uint_least8_t

21.5.2.40 PRIuPTR `#define PRIuPTR PRIu16`

decimal printf format for uintptr_t

21.5.2.41 PRIx16 `#define PRIx16 "x"`

hexadecimal printf format for uint16_t

21.5.2.42 PRIX16 `#define PRIX16 "X"`

uppercase hexadecimal printf format for uint16_t

21.5.2.43 PRIx32 `#define PRIx32 "lx"`

hexadecimal printf format for uint32_t

21.5.2.44 PRIX32 `#define PRIX32 "lX"`

uppercase hexadecimal printf format for uint32_t

21.5.2.45 PRIx8 `#define PRIx8 "x"`

hexadecimal printf format for uint8_t

21.5.2.46 PRIX8 `#define PRIX8 "X"`

uppercase hexadecimal printf format for uint8_t

21.5.2.47 PRIxFAST16 `#define PRIxFAST16 "x"`

hexadecimal printf format for uint_fast16_t

21.5.2.48 PRIXFAST16 `#define PRIXFAST16 "X"`

uppercase hexadecimal printf format for uint_fast16_t

21.5.2.49 PRIxFAST32 `#define PRIxFAST32 "lx"`

hexadecimal printf format for uint_fast32_t

21.5.2.50 PRIXFAST32 `#define PRIXFAST32 "lX"`

uppercase hexadecimal printf format for uint_fast32_t

21.5.2.51 PRIxFAST8 `#define PRIxFAST8 "x"`

hexadecimal printf format for uint_fast8_t

21.5.2.52 PRIXFAST8 `#define PRIXFAST8 "X"`

uppercase hexadecimal printf format for uint_fast8_t

21.5.2.53 PRIxLEAST16 `#define PRIxLEAST16 "x"`

hexadecimal printf format for uint_least16_t

21.5.2.54 PRIXLEAST16 `#define PRIXLEAST16 "X"`

uppercase hexadecimal printf format for uint_least16_t

21.5.2.55 PRIxLEAST32 `#define PRIxLEAST32 "lx"`

hexadecimal printf format for uint_least32_t

21.5.2.56 PRIXLEAST32 `#define PRIXLEAST32 "lX"`

uppercase hexadecimal printf format for uint_least32_t

21.5.2.57 PRIxLEAST8 `#define PRIxLEAST8 "x"`

hexadecimal printf format for uint_least8_t

21.5.2.58 PRIXLEAST8 `#define PRIXLEAST8 "X"`

uppercase hexadecimal printf format for uint_least8_t

21.5.2.59 PRIxPTR `#define PRIxPTR PRIx16`

hexadecimal printf format for uintptr_t

21.5.2.60 PRIXPTR `#define PRIXPTR PRIX16`

uppercase hexadecimal printf format for uintptr_t

21.5.2.61 SCNd16 `#define SCNd16 "d"`

decimal scanf format for int16_t

21.5.2.62 SCNd32 `#define SCNd32 "ld"`

decimal scanf format for int32_t

21.5.2.63 SCNd8 `#define SCNd8 "hhd"`

decimal scanf format for int8_t

21.5.2.64 SCNdFAST16 `#define SCNdFAST16 "d"`

decimal scanf format for int_fast16_t

21.5.2.65 SCNdFAST32 `#define SCNdFAST32 "ld"`

decimal scanf format for int_fast32_t

21.5.2.66 SCNdFAST8 `#define SCNdFAST8 "hhd"`

decimal scanf format for int_fast8_t

21.5.2.67 SCNdLEAST16 `#define SCNdLEAST16 "d"`

decimal scanf format for int_least16_t

21.5.2.68 SCNdLEAST32 `#define SCNdLEAST32 "ld"`

decimal scanf format for int_least32_t

21.5.2.69 SCNdLEAST8 `#define SCNdLEAST8 "hhd"`

decimal scanf format for int_least8_t

21.5.2.70 SCNdPTR `#define SCNdPTR SCNd16`

decimal scanf format for intptr_t

21.5.2.71 SCNi16 `#define SCNi16 "i"`

generic-integer scanf format for int16_t

21.5.2.72 SCNi32 `#define SCNi32 "li"`

generic-integer scanf format for int32_t

21.5.2.73 SCNi8 `#define SCNi8 "hhi"`

generic-integer scanf format for int8_t

21.5.2.74 SCNiFAST16 `#define SCNiFAST16 "i"`

generic-integer scanf format for int_fast16_t

21.5.2.75 SCNiFAST32 `#define SCNiFAST32 "li"`

generic-integer scanf format for int_fast32_t

21.5.2.76 SCNiFAST8 `#define SCNiFAST8 "hhi"`

generic-integer scanf format for int_fast8_t

21.5.2.77 SCNiLEAST16 `#define SCNiLEAST16 "i"`

generic-integer scanf format for int_least16_t

21.5.2.78 SCNiLEAST32 `#define SCNiLEAST32 "li"`

generic-integer scanf format for int_least32_t

21.5.2.79 SCNiLEAST8 `#define SCNiLEAST8 "hhi"`

generic-integer scanf format for int_least8_t

21.5.2.80 SCNiPTR `#define SCNiPTR SCNi16`

generic-integer scanf format for intptr_t

21.5.2.81 SCNo16 `#define SCNo16 "o"`

octal scanf format for uint16_t

21.5.2.82 SCNo32 `#define SCNo32 "lo"`

octal scanf format for uint32_t

21.5.2.83 SCNo8 `#define SCNo8 "hho"`

octal scanf format for uint8_t

21.5.2.84 SCNoFAST16 `#define SCNoFAST16 "o"`

octal scanf format for uint_fast16_t

21.5.2.85 SCNoFAST32 `#define SCNoFAST32 "lo"`

octal scanf format for uint_fast32_t

21.5.2.86 SCNoFAST8 `#define SCNoFAST8 "hho"`

octal scanf format for uint_fast8_t

21.5.2.87 SCNoLEAST16 `#define SCNoLEAST16 "o"`

octal scanf format for uint_least16_t

21.5.2.88 SCNoLEAST32 `#define SCNoLEAST32 "lo"`

octal scanf format for uint_least32_t

21.5.2.89 SCNoLEAST8 `#define SCNoLEAST8 "hho"`

octal scanf format for uint_least8_t

21.5.2.90 SCNoPTR `#define SCNoPTR SCNo16`

octal scanf format for uintptr_t

21.5.2.91 SCNu16 `#define SCNu16 "u"`

decimal scanf format for uint16_t

21.5.2.92 SCNu32 `#define SCNu32 "lu"`

decimal scanf format for uint32_t

21.5.2.93 SCNu8 `#define SCNu8 "hhu"`

decimal scanf format for uint8_t

21.5.2.94 SCNuFAST16 `#define SCNuFAST16 "u"`

decimal scanf format for uint_fast16_t

21.5.2.95 SCNuFAST32 `#define SCNuFAST32 "lu"`

decimal scanf format for uint_fast32_t

21.5.2.96 SCNuFAST8 `#define SCNuFAST8 "hhu"`

decimal scanf format for uint_fast8_t

21.5.2.97 SCNuLEAST16 `#define SCNuLEAST16 "u"`

decimal scanf format for uint_least16_t

21.5.2.98 SCNuLEAST32 `#define SCNuLEAST32 "lu"`

decimal scanf format for uint_least32_t

21.5.2.99 SCNuLEAST8 `#define SCNuLEAST8 "hhu"`

decimal scanf format for uint_least8_t

21.5.2.100 SCNuPTR `#define SCNuPTR SCNu16`

decimal scanf format for uintptr_t

21.5.2.101 SCNx16 `#define SCNx16 "x"`

hexadecimal scanf format for uint16_t

21.5.2.102 SCNx32 `#define SCNx32 "lx"`

hexadecimal scanf format for uint32_t

21.5.2.103 SCNx8 `#define SCNx8 "hhx"`

hexadecimal scanf format for uint8_t

21.5.2.104 SCNxFAST16 `#define SCNxFAST16 "x"`

hexadecimal scanf format for uint_fast16_t

21.5.2.105 SCNxFAST32 `#define SCNxFAST32 "lx"`

hexadecimal scanf format for uint_fast32_t

21.5.2.106 SCNxFAST8 `#define SCNxFAST8 "hhx"`

hexadecimal scanf format for uint_fast8_t

21.5.2.107 SCNxLEAST16 `#define SCNxLEAST16 "x"`

hexadecimal scanf format for uint_least16_t

21.5.2.108 SCNxLEAST32 `#define SCNxLEAST32 "lx"`

hexadecimal scanf format for uint_least32_t

21.5.2.109 SCNxLEAST8 `#define SCNxLEAST8 "hhx"`

hexadecimal scanf format for uint_least8_t

21.5.2.110 SCNxPTR `#define SCNxPTR SCNx16`

hexadecimal scanf format for uintptr_t

21.5.3 Typedef Documentation

21.5.3.1 int_farptr_t `typedef int32_t int_farptr_t`

signed integer type that can hold a pointer > 64 KiB

21.5.3.2 uint_farptr_t `typedef uint32_t uint_farptr_t`

unsigned integer type that can hold a pointer > 64 KiB, see also pgm_get_far_address()

21.6 <math.h>: Mathematics

Macros

- #define M_E 2.7182818284590452354
- #define M_LOG2E 1.4426950408889634074
- #define M_LOG10E 0.43429448190325182765
- #define M_LN2 0.69314718055994530942
- #define M_LN10 2.30258509299404568402
- #define M_PI 3.14159265358979323846
- #define M_PI_2 1.57079632679489661923
- #define M_PI_4 0.78539816339744830962
- #define M_1_PI 0.31830988618379067154
- #define M_2_PI 0.63661977236758134308
- #define M_2_SQRTPI 1.12837916709551257390
- #define M_SQRT2 1.41421356237309504880
- #define M_SQRT1_2 0.70710678118654752440
- #define NAN __builtin_nan("")
- #define nanf(__tagp) __builtin_nanf(__tag)
- #define nan(__tag) __builtin_nan(__tag)
- #define nanl(__tag) __builtin_nanl(__tag)
- #define INFINITY __builtin_inf()
- #define HUGE_VALF __builtin_huge_valf()
- #define HUGE_VAL __builtin_huge_val()
- #define HUGE_VALL __builtin_huge_vall()

Functions

- float cosf (float x)
- double cos (double x)
- long double cosl (long double x)
- float sinf (float x)
- double sin (double x)
- long double sinl (long double x)
- float tanf (float x)
- double tan (double x)
- long double tanl (long double x)
- static float fabsf (float __x)
- static double fabs (double __x)
- static long double fabsl (long double __x)
- float fmodf (float x, float y)
- double fmod (double x, double y)
- long double fmodl (long double x, long double y)
- float modff (float x, float ∗iptr)
- double modf (double x, double ∗iptr)
- long double modfl (long double x, long double ∗iptr)
- float sqrtf (float x)
- double sqrt (double x)
- long double sqrtl (long double x)
- float cbrtf (float x)
- double cbrt (double x)
- long double cbrtl (long double x)
- float hypotf (float x, float y)

- double hypot (double x, double y)
- long double hypotl (long double x, long double y)
- float floorf (float x)
- double floor (double x)
- long double floorl (long double x)
- float ceilf (float x)
- double ceil (double x)
- long double ceill (long double x)
- float frexpf (float x, int ∗pexp)
- double frexp (double x, int ∗pexp)
- long double frexpl (long double x, int ∗pexp)
- float ldexpf (float x, int iexp)
- double ldexp (double x, int iexp)
- long double ldexpl (long double x, int iexp)
- float expf (float x)
- double exp (double x)
- long double expl (long double x)
- float coshf (float x)
- double cosh (double x)
- long double coshl (long double x)
- float sinhf (float x)
- double sinh (double x)
- long double sinhl (long double x)
- float tanhf (float x)
- double tanh (double x)
- long double tanhl (long double x)
- float acosf (float x)
- double acos (double x)
- long double acosl (long double x)
- float asinf (float x)
- double asin (double x)
- long double asinl (long double x)
- float atanf (float x)
- double atan (double x)
- long double atanl (long double x)
- float atan2f (float y, float x)
- double atan2 (double y, double x)
- long double atan2l (long double y, long double x)
- float logf (float x)
- double log (double x)
- long double logl (long double x)
- float log10f (float x)
- double log10 (double x)
- long double log10l (long double x)
- float powf (float x, float y)
- double pow (double x, double y)
- long double powl (long double x, long double y)
- int isnanf (float x)
- int isnan (double x)
- int isnanl (long double x)
- int isinff (float x)
- int isinf (double x)
- int isinfl (long double x)
- static int isfinitef (float __x)
- static int isfinite (double __x)

- static int isfinitel (long double __x)
- static float copysignf (float __x, float __y)
- static double copysign (double __x, double __y)
- static long double copysignl (long double __x, long double __y)
- int signbitf (float x)
- int signbit (double x)
- int signbitl (long double x)
- float fdimf (float x, float y)
- double fdim (double x, double y)
- long double fdiml (long double x, long double y)
- float fmaf (float x, float y, float z)
- double fma (double x, double y, double z)
- long double fmal (long double x, long double y, long double z)
- float fmaxf (float x, float y)
- double fmax (double x, double y)
- long double fmaxl (long double x, long double y)
- float fminf (float x, float y)
- double fmin (double x, double y)
- long double fminl (long double x, long double y)
- float truncf (float x)
- double trunc (double x)
- long double truncl (long double x)
- float roundf (float x)
- double round (double x)
- long double roundl (long double x)
- long lroundf (float x)
- long lround (double x)
- long lroundl (long double x)
- long lrintf (float x)
- long lrint (double x)
- long lrintl (long double x)

Non-Standard Math Functions

- float squaref (float x)
- double square (double x)
- long double squarel (long double x)

21.6.1 Detailed Description

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

This header file declares basic mathematics constants and functions.

Notes:

- Math functions do not raise exceptions and do not change the `errno` variable. Therefore the majority of them are declared with `const` attribute, for better optimization by GCC.
- 64-bit floating-point arithmetic is only available in `avr-gcc v10` and up. The size of the `double` and `long double` type can be selected at compile-time with options like `-mdouble=64` and `-mlong-double=32`. Whether such options are available, and their default values, depend on how the compiler has been configured.
- The implementation of 64-bit floating-point arithmetic has some shortcomings and limitations, see the `avr-gcc Wiki` for details.
- In order to access the `float` functions, in avr-gcc v4.6 and older it is usually also required to link with `-lm`. In avr-gcc v4.7 and up, `-lm` is added automatically to all linker invocations.

21.6.2 Macro Definition Documentation

21.6.2.1 HUGE_VAL `#define HUGE_VAL __builtin_huge_val()`

`double` infinity constant.

21.6.2.2 HUGE_VALF `#define HUGE_VALF __builtin_huge_valf()`

`float` infinity constant.

21.6.2.3 HUGE_VALL `#define HUGE_VALL __builtin_huge_vall()`

`long double` infinity constant.

21.6.2.4 INFINITY `#define INFINITY __builtin_inf()`

`double` infinity constant.

21.6.2.5 M_1_PI `#define M_1_PI 0.31830988618379067154`

The constant *1/pi*.

21.6.2.6 M_2_PI `#define M_2_PI 0.63661977236758134308`

The constant *2/pi*.

21.6.2.7 M_2_SQRTPI `#define M_2_SQRTPI 1.12837916709551257390`

The constant *2/sqrt*(pi).

21.6.2.8 M_E `#define M_E 2.7182818284590452354`

The constant Euler's number *e*.

21.6.2.9 M_LN10 `#define M_LN10 2.30258509299404568402`

The constant natural logarithm of 10.

21.6.2.10 M_LN2 `#define M_LN2 0.69314718055994530942`

The constant natural logarithm of 2.

21.6.2.11 M_LOG10E `#define M_LOG10E 0.43429448190325182765`

The constant logarithm of Euler's number *e* to base 10.

21.6.2.12 M_LOG2E `#define M_LOG2E 1.4426950408889634074`

The constant logarithm of Euler's number *e* to base 2.

21.6.2.13 M_PI `#define M_PI 3.14159265358979323846`

The constant *pi*.

21.6.2.14 M_PI_2 `#define M_PI_2 1.57079632679489661923`

The constant *pi/2*.

21.6.2.15 M_PI_4 `#define M_PI_4 0.78539816339744830962`

The constant *pi/4*.

21.6.2.16 M_SQRT1_2 `#define M_SQRT1_2 0.70710678118654752440`

The constant *1/sqrt*(2).

21.6.2.17 M_SQRT2 `#define M_SQRT2 1.41421356237309504880`

The square root of 2.

21.6.2.18 NAN `#define NAN __builtin_nan("")`

The `double` representation of a constant quiet NaN.

21.6.2.19 nan `#define nan( __tag ) __builtin_nan(__tag)`

The `double` representation of a constant quiet NaN. `__tag` is a string constant like `""` or `"123"`.

21.6.2.20 nanf `#define nanf( __tagp ) __builtin_nanf(__tag)`

The `float` representation of a constant quiet NaN. `__tag` is a string constant like `""` or `"123"`.

21.6.2.21 nanl `#define nanl( __tag ) __builtin_nanl(__tag)`

The `long double` representation of a constant quiet NaN. `__tag` is a string constant like `""` or `"123"`.

21.6.3 Function Documentation

21.6.3.1 acos()

```
double acos (
    double x )
```

The acos() function computes the principal value of the arc cosine of *x*. The returned value is in the range [0, pi] radians or NaN.

21.6.3.2 acosf()

```
float acosf (
    float x )
```

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

21.6.3.3 acosl()

```
long double acosl (
    long double x )
```

The acosl() function computes the principal value of the arc cosine of *x*. The returned value is in the range [0, pi] radians or NaN.

21.6.3.4 asin()

```
double asin (
    double x )
```

The asin() function computes the principal value of the arc sine of *x*. The returned value is in the range [-pi/2, pi/2] radians or NaN.

21.6.3.5 asinf()

```
float asinf (
    float x )
```

The asinf() function computes the principal value of the arc sine of *x*. The returned value is in the range [-pi/2, pi/2] radians. A domain error occurs for arguments not in the range [-1, +1].

21.6.3.6 asinl()

```
long double asinl (
    long double x )
```

The asinl() function computes the principal value of the arc sine of *x*. The returned value is in the range [-pi/2, pi/2] radians or NaN.

21.6.3.7 atan()

```
double atan (
    double x )
```

The atan() function computes the principal value of the arc tangent of *x*. The returned value is in the range [-pi/2, pi/2] radians.

21.6.3.8 atan2()

```
double atan2 (
    double y,
    double x )
```

The atan2() function computes the principal value of the arc tangent of *y / x*, using the signs of both arguments to determine the quadrant of the return value. The returned value is in the range [-pi, +pi] radians.

21.6.3.9 atan2f()

```
float atan2f (
    float y,
    float x )
```

The atan2f() function computes the principal value of the arc tangent of *y / x*, using the signs of both arguments to determine the quadrant of the return value. The returned value is in the range [-pi, +pi] radians.

21.6.3.10 atan2l()

```
long double atan2l (
    long double y,
    long double x )
```

The atan2l() function computes the principal value of the arc tangent of *y / x*, using the signs of both arguments to determine the quadrant of the return value. The returned value is in the range [-pi, +pi] radians.

21.6.3.11 atanf()

```
float atanf (
    float x )
```

The atanf() function computes the principal value of the arc tangent of *x*. The returned value is in the range [-pi/2, pi/2] radians.

21.6.3.12 atanl()

```
long double atanl (
    long double x )
```

The atanl() function computes the principal value of the arc tangent of *x*. The returned value is in the range [-pi/2, pi/2] radians.

21.6.3.13 cbrt()

```
double cbrt (
    double x )
```

The cbrt() function returns the cube root of *x*.

21.6.3.14 cbrtf()

```
float cbrtf (
    float x )
```

The cbrtf() function returns the cube root of *x*.

21.6.3.15 cbrtl()

```
long double cbrtl (
    long double x )
```

The cbrtl() function returns the cube root of *x*.

21.6.3.16 ceil()

```
double ceil (
    double x )
```

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

21.6.3.17 ceilf()

```
float ceilf (
    float x )
```

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

21.6.3.18 ceill()

```
long double ceill (
    long double x )
```

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

21.6.3.19 copysign()

```
static double copysign (
    double __x,
    double __y ) [inline], [static]
```

The copysign() function returns *__x* but with the sign of *__y*. They work even if *__x* or *__y* are NaN or zero.

21.6.3.20 copysignf()

```
static float copysignf (
    float __x,
    float __y ) [inline], [static]
```

The copysignf() function returns *__x* but with the sign of *__y*. They work even if *__x* or *__y* are NaN or zero.

21.6.3.21 copysignl()

```
static long double copysignl (
    long double __x,
    long double __y ) [inline], [static]
```

The copysignl() function returns *__x* but with the sign of *__y*. They work even if *__x* or *__y* are NaN or zero.

21.6.3.22 cos()

```
double cos (
    double x )
```

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

21.6.3.23 cosf()

```
float cosf (
    float x )
```

The cosf() function returns the cosine of *x*, measured in radians.

21.6.3.24 cosh()

```
double cosh (
    double x )
```

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

21.6.3.25 coshf()

```
float coshf (
    float x )
```

The coshf() function returns the hyperbolic cosine of *x*.

21.6.3.26 coshl()

```
long double coshl (
    long double x )
```

The coshl() function returns the hyperbolic cosine of *x*.

21.6.3.27 cosl()

```
long double cosl (
    long double x )
```

The cosl() function returns the cosine of *x*, measured in radians.

21.6.3.28 exp()

```
double exp (
    double x )
```

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

21.6.3.29 expf()

```
float expf (
    float x )
```

The expf() function returns the exponential value of *x*.

21.6.3.30 expl()

```
long double expl (
    long double x )
```

The expl() function returns the exponential value of *x*.

21.6.3.31 fabs()

```
static double fabs (
    double __x ) [inline], [static]
```

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

21.6.3.32 fabsf()

```
static float fabsf (
    float __x ) [inline], [static]
```

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

21.6.3.33 fabsl()

```
static long double fabsl (
    long double __x ) [inline], [static]
```

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

21.6.3.34 fdim() `double fdim ( double x, double y )`

The fdim() function returns *max(x - y, 0)*. If *x* or *y* or both are NaN, NaN is returned.

21.6.3.35 fdimf() `float fdimf ( float x, float y )`

The fdimf() function returns *max(x - y, 0)*. If *x* or *y* or both are NaN, NaN is returned.

21.6.3.36 fdiml() `long double fdiml ( long double x, long double y )`

The fdiml() function returns *max(x - y, 0)*. If *x* or *y* or both are NaN, NaN is returned.

21.6.3.37 floor() `double floor ( double x )`

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

21.6.3.38 floorf() `float floorf ( float x )`

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

21.6.3.39 floorl() `long double floorl ( long double x )`

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

21.6.3.40 fma() `double fma ( double x, double y, double z )`

The fma() function performs floating-point multiply-add. This is the operation $(x * y) + z$, but the intermediate result is not rounded to the destination type. This can sometimes improve the precision of a calculation.

21.6.3.41 fmaf() `float fmaf ( float x, float y, float z )`

The fmaf() function performs floating-point multiply-add. This is the operation $(x * y) + z$, but the intermediate result is not rounded to the destination type. This can sometimes improve the precision of a calculation.

21.6.3.42 fmal()

```
long double fmal (
    long double x,
    long double y,
    long double z )
```

The fmal() function performs floating-point multiply-add. This is the operation *(x ∗ y) + z*, but the intermediate result is not rounded to the destination type. This can sometimes improve the precision of a calculation.

21.6.3.43 fmax()

```
double fmax (
    double x,
    double y )
```

The fmax() function returns the greater of the two values *x* and *y*. If an argument is NaN, the other argument is returned. If both arguments are NaN, NaN is returned.

21.6.3.44 fmaxf()

```
float fmaxf (
    float x,
    float y )
```

The fmaxf() function returns the greater of the two values *x* and *y*. If an argument is NaN, the other argument is returned. If both arguments are NaN, NaN is returned.

21.6.3.45 fmaxl()

```
long double fmaxl (
    long double x,
    long double y )
```

The fmaxl() function returns the greater of the two values *x* and *y*. If an argument is NaN, the other argument is returned. If both arguments are NaN, NaN is returned.

21.6.3.46 fmin()

```
double fmin (
    double x,
    double y )
```

The fmin() function returns the lesser of the two values *x* and *y*. If an argument is NaN, the other argument is returned. If both arguments are NaN, NaN is returned.

21.6.3.47 fminf()

```
float fminf (
    float x,
    float y )
```

The fminf() function returns the lesser of the two values *x* and *y*. If an argument is NaN, the other argument is returned. If both arguments are NaN, NaN is returned.

21.6.3.48 fminl()

```
long double fminl (
    long double x,
    long double y )
```

The fminl() function returns the lesser of the two values *x* and *y*. If an argument is NaN, the other argument is returned. If both arguments are NaN, NaN is returned.

21.6.3.49 fmod()

```
double fmod (
    double x,
    double y )
```

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

21.6.3.50 fmodf()

```
float fmodf (
    float x,
    float y )
```

The function fmodf() returns the floating-point remainder of *x / y*.

21.6.3.51 fmodl()

```
long double fmodl (
    long double x,
    long double y )
```

The function fmodl() returns the floating-point remainder of *x / y*.

21.6.3.52 frexp()

```
double frexp (
    double x,
    int * pexp )
```

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

If *x* is a normal float point number, the frexp() function returns the value `v`, such that `v` has a magnitude in the interval [1/2, 1) or zero, and *x* equals `v` times 2 raised to the power *pexp*. If *x* is zero, both parts of the result are zero. If *x* is not a finite number, the frexp() returns *x* as is and stores 0 by *pexp*.

21.6.3.53 frexpf()

```
float frexpf (
    float x,
    int * pexp )
```

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

If *x* is a normal float point number, the frexpf() function returns the value `v`, such that `v` has a magnitude in the interval [1/2, 1) or zero, and *x* equals `v` times 2 raised to the power *pexp*. If *x* is zero, both parts of the result are zero. If *x* is not a finite number, the frexpf() returns *x* as is and stores 0 by *pexp*.

Note

This implementation permits a zero pointer as a directive to skip a storing the exponent.

21.6.3.54 frexpl()

```
long double frexpl (
    long double x,
    int * pexp )
```

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

If *x* is a normal float point number, the frexpl() function returns the value `v`, such that `v` has a magnitude in the interval [1/2, 1) or zero, and *x* equals `v` times 2 raised to the power *pexp*. If *x* is zero, both parts of the result are zero. If *x* is not a finite number, the frexpl() returns *x* as is and stores 0 by *pexp*.

21.6.3.55 hypot()

```
double hypot (
    double x,
    double y )
```

The hypot() function returns *sqrt(x*x + y*y)*. This is the length of the hypotenuse of a right triangle with sides of length *x* and *y*, or the distance of the point (*x*, *y*) from the origin. Using this function instead of the direct formula is wise, since the error is much smaller. No underflow with small *x* and *y*. No overflow if result is in range.

21.6.3.56 hypotf()

```
float hypotf (
    float x,
    float y )
```

The hypotf() function returns *sqrtf(x*x + y*y)*. This is the length of the hypotenuse of a right triangle with sides of length *x* and *y*, or the distance of the point (*x*, *y*) from the origin. Using this function instead of the direct formula is wise, since the error is much smaller. No underflow with small *x* and *y*. No overflow if result is in range.

21.6.3.57 hypotl()

```
long double hypotl (
    long double x,
    long double y )
```

The hypotl() function returns *sqrtl(x*x + y*y)*. This is the length of the hypotenuse of a right triangle with sides of length *x* and *y*, or the distance of the point (*x*, *y*) from the origin. Using this function instead of the direct formula is wise, since the error is much smaller. No underflow with small *x* and *y*. No overflow if result is in range.

21.6.3.58 isfinite()

```
static int isfinite (
    double __x ) [inline], [static]
```

The isfinite() function returns a nonzero value if *__x* is finite: not plus or minus infinity, and not NaN.

21.6.3.59 isfinitef()

```
static int isfinitef (
    float __x ) [inline], [static]
```

The isfinitef() function returns a nonzero value if *__x* is finite: not plus or minus infinity, and not NaN.

21.6.3.60 isfinitel()

```
static int isfinitel (
    long double __x ) [inline], [static]
```

The isfinite() function returns a nonzero value if *__x* is finite: not plus or minus infinity, and not NaN.

21.6.3.61 isinf()

```
int isinf (
    double x )
```

The function isinf() returns 1 if the argument *x* is positive infinity, -1 if *x* is negative infinity, and 0 otherwise.

21.6.3.62 isinff()

```
int isinff (
    float x )
```

The function isinff() returns 1 if the argument *x* is positive infinity, -1 if *x* is negative infinity, and 0 otherwise.

21.6.3.63 isinfl() `int isinfl ( long double x )`

The function isinfl() returns 1 if the argument *x* is positive infinity, -1 if *x* is negative infinity, and 0 otherwise.

21.6.3.64 isnan() `int isnan ( double x )`

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

21.6.3.65 isnanf() `int isnanf ( float x )`

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

21.6.3.66 isnanl() `int isnanl ( long double x )`

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

21.6.3.67 ldexp() `double ldexp ( double x, int iexp )`

The ldexp() function multiplies a floating-point number by an integral power of 2. It returns the value of *x* times 2 raised to the power *iexp*.

21.6.3.68 ldexpf() `float ldexpf ( float x, int iexp )`

The ldexpf() function multiplies a floating-point number by an integral power of 2. It returns the value of *x* times 2 raised to the power *iexp*.

21.6.3.69 ldexpl() `long double ldexpl ( long double x, int iexp )`

The ldexpl() function multiplies a floating-point number by an integral power of 2. It returns the value of *x* times 2 raised to the power *iexp*.

21.6.3.70 log() `double log ( double x )`

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

21.6.3.71 log10() `double log10 ( double x )`

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

21.6.3.72 log10f()

```
float log10f (
    float x )
```

The log10f() function returns the logarithm of argument *x* to base 10.

21.6.3.73 log10l()

```
long double log10l (
    long double x )
```

The log10l() function returns the logarithm of argument *x* to base 10.

21.6.3.74 logf()

```
float logf (
    float x )
```

The logf() function returns the natural logarithm of argument *x*.

21.6.3.75 logl()

```
long double logl (
    long double x )
```

The logl() function returns the natural logarithm of argument *x*.

21.6.3.76 lrint()

```
long lrint (
    double x )
```

The lrint() function rounds *x* to the nearest integer, rounding the halfway cases to the even integer direction. (That is both 1.5 and 2.5 values are rounded to 2). This function is similar to rint() function, but it differs in type of return value and in that an overflow is possible.

Returns

The rounded long integer value. If *x* is not a finite number or an overflow was, this realization returns the `LONG_MIN` value (0x80000000).

21.6.3.77 lrintf()

```
long lrintf (
    float x )
```

The lrintf() function rounds *x* to the nearest integer, rounding the halfway cases to the even integer direction. (That is both 1.5 and 2.5 values are rounded to 2). This function is similar to rintf() function, but it differs in type of return value and in that an overflow is possible.

Returns

The rounded long integer value. If *x* is not a finite number or an overflow was, this realization returns the `LONG_MIN` value (0x80000000).

21.6.3.78 lrintl() `long lrintl ( long double x )`

The lrintl() function rounds *x* to the nearest integer, rounding the halfway cases to the even integer direction. (That is both 1.5 and 2.5 values are rounded to 2). This function is similar to rintl() function, but it differs in type of return value and in that an overflow is possible.

Returns

The rounded long integer value. If *x* is not a finite number or an overflow was, this realization returns the `LONG_MIN` value (0x80000000).

21.6.3.79 lround() `long lround ( double x )`

The lround() function rounds *x* to the nearest integer, but rounds halfway cases away from zero (instead of to the nearest even integer). This function is similar to round() function, but it differs in type of return value and in that an overflow is possible.

Returns

The rounded long integer value. If *x* is not a finite number or an overflow was, this realization returns the `LONG_MIN` value (0x80000000).

21.6.3.80 lroundf() `long lroundf ( float x )`

The lroundf() function rounds *x* to the nearest integer, but rounds halfway cases away from zero (instead of to the nearest even integer). This function is similar to round() function, but it differs in type of return value and in that an overflow is possible.

Returns

The rounded long integer value. If *x* is not a finite number or an overflow was, this realization returns the `LONG_MIN` value (0x80000000).

21.6.3.81 lroundl() `long lroundl ( long double x )`

The lroundl() function rounds *x* to the nearest integer, but rounds halfway cases away from zero (instead of to the nearest even integer). This function is similar to round() function, but it differs in type of return value and in that an overflow is possible.

Returns

The rounded long integer value. If *x* is not a finite number or an overflow was, this realization returns the `LONG_MIN` value (0x80000000).

21.6.3.82 modf()

```
double modf (
    double x,
    double * iptr )
```

The modf() function breaks the argument *x* into integral and fractional parts, each of which has the same sign as the argument. It stores the integral part as a `double` in the object pointed to by *iptr*.

The modf() function returns the signed fractional part of *x*.

21.6.3.83 modff()

```
float modff (
    float x,
    float * iptr )
```

The modff() function breaks the argument *x* into integral and fractional parts, each of which has the same sign as the argument. It stores the integral part as a `float` in the object pointed to by *iptr*.

The modff() function returns the signed fractional part of *x*.

Note

This implementation skips writing by zero pointer. However, the GCC 4.3 can replace this function with inline code that does not permit to use NULL address for the avoiding of storing.

21.6.3.84 modfl()

```
long double modfl (
    long double x,
    long double * iptr )
```

The modfl() function breaks the argument *x* into integral and fractional parts, each of which has the same sign as the argument. It stores the integral part as a `long double` in the object pointed to by *iptr*.

The modf() function returns the signed fractional part of *x*.

21.6.3.85 pow()

```
double pow (
    double x,
    double y )
```

The function pow() returns the value of *x* to the exponent *y*.
Notice that for integer exponents, there is the more efficient `double __builtin_powi(double x, int y)`.

21.6.3.86 powf()

```
float powf (
    float x,
    float y )
```

The function powf() returns the value of *x* to the exponent *y*.
Notice that for integer exponents, there is the more efficient `float __builtin_powif(float x, int y)`.

21.6.3.87 powl()

```
long double powl (
    long double x,
    long double y )
```

The function powl() returns the value of *x* to the exponent *y*.
Notice that for integer exponents, there is the more efficient `long double __builtin_powil(long double x, int y)`.

21.6.3.88 round()

```
double round (
    double x )
```

The round() function rounds *x* to the nearest integer, but rounds halfway cases away from zero (instead of to the nearest even integer). Overflow is impossible.

Returns

The rounded value. If *x* is an integral or infinite, *x* itself is returned. If *x* is `NaN`, then `NaN` is returned.

21.6.3.89 roundf()

```
float roundf (
    float x )
```

The roundf() function rounds *x* to the nearest integer, but rounds halfway cases away from zero (instead of to the nearest even integer). Overflow is impossible.

Returns

The rounded value. If *x* is an integral or infinite, *x* itself is returned. If *x* is `NaN`, then `NaN` is returned.

21.6.3.90 roundl()

```
long double roundl (
    long double x )
```

The roundl() function rounds *x* to the nearest integer, but rounds halfway cases away from zero (instead of to the nearest even integer). Overflow is impossible.

Returns

The rounded value. If *x* is an integral or infinite, *x* itself is returned. If *x* is `NaN`, then `NaN` is returned.

21.6.3.91 signbit()

```
int signbit (
    double x )
```

The signbit() function returns a nonzero value if the value of *x* has its sign bit set. This is not the same as `$x < 0.0$', because IEEE 754 floating point allows zero to be signed. The comparison '-0.0 < 0.0' is false, but `signbit (-0.0)' will return a nonzero value.

21.6.3.92 signbitf()

```
int signbitf (
    float x )
```

The signbitf() function returns a nonzero value if the value of *x* has its sign bit set. This is not the same as `$x < 0.0$', because IEEE 754 floating point allows zero to be signed. The comparison '-0.0 $<$ 0.0' is false, but `signbit (-0.0)' will return a nonzero value.

21.6.3.93 signbitl()

```
int signbitl (
    long double x )
```

The signbitl() function returns a nonzero value if the value of *x* has its sign bit set. This is not the same as `$x < 0.0$', because IEEE 754 floating point allows zero to be signed. The comparison '-0.0 $<$ 0.0' is false, but `signbit (-0.0)' will return a nonzero value.

21.6.3.94 sin()

```
double sin (
    double x )
```

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

21.6.3.95 sinf()

```
float sinf (
    float x )
```

The sinf() function returns the sine of *x*, measured in radians.

21.6.3.96 sinh()

```
double sinh (
    double x )
```

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

21.6.3.97 sinhf()

```
float sinhf (
    float x )
```

The sinhf() function returns the hyperbolic sine of *x*.

21.6.3.98 sinhl()

```
long double sinhl (
    long double x )
```

The sinhl() function returns the hyperbolic sine of *x*.

21.6.3.99 sinl()

```
long double sinl (
    long double x )
```

The sinl() function returns the sine of *x*, measured in radians.

21.6.3.100 sqrt()

```
double sqrt (
    double x )
```

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

21.6.3.101 sqrtf()

```
float sqrtf (
    float x )
```

The sqrtf() function returns the non-negative square root of *x*.

21.6.3.102 sqrtl()

```
long double sqrtl (
    long double x )
```

The sqrtl() function returns the non-negative square root of *x*.

21.6.3.103 square()

```
double square (
    double x )
```

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

Note

This function does not belong to the C standard definition.

21.6.3.104 squaref()

```
float squaref (
    float x )
```

The function squaref() returns $x * x$.

Note

This function does not belong to the C standard definition.

21.6.3.105 squarel()

```
long double squarel (
    long double x )
```

The function squarel() returns $x * x$.

Note

This function does not belong to the C standard definition.

21.6.3.106 tan()

```
double tan (
    double x )
```

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

21.6.3.107 tanf()

```
float tanf (
    float x )
```

The tanf() function returns the tangent of *x*, measured in radians.

21.6.3.108 tanh()

```
double tanh (
    double x )
```

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

21.6.3.109 tanhf()

```
float tanhf (
    float x )
```

The tanhf() function returns the hyperbolic tangent of *x*.

21.6.3.110 tanhl()

```
long double tanhl (
    long double x )
```

The tanhl() function returns the hyperbolic tangent of *x*.

21.6.3.111 tanl()

```
long double tanl (
    long double x )
```

The tanl() function returns the tangent of *x*, measured in radians.

21.6.3.112 trunc()

```
double trunc (
    double x )
```

The trunc() function rounds *x* to the nearest integer not larger in absolute value.

21.6.3.113 truncf()

```
float truncf (
    float x )
```

The truncf() function rounds *x* to the nearest integer not larger in absolute value.

21.6.3.114 truncl()

```
long double truncl (
    long double x )
```

The truncl() function rounds *x* to the nearest integer not larger in absolute value.

21.7 <setjmp.h>: Non-local goto

Functions

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

21.7.1 Detailed Description

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

Note

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

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

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

Example:

```
#include <setjmp.h>

jmp_buf env;

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

    while (1)
    {
       // Main processing loop which calls foo() somewhere ...
    }
}

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

    if (err)
    {
        longjmp (env, 1);
    }
}
```

21.7.2 Function Documentation

21.7.2.1 longjmp() `void longjmp ( jmp_buf __jmpb, int __ret )`

Non-local jump to a saved stack context.

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

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

Note

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

Parameters

__jmpb	Information saved by a previous call to setjmp().
__ret	Value to return to the caller of setjmp().

Returns

This function never returns.

21.7.2.2 setjmp() `int setjmp ( jmp_buf __jmpb )`

Save stack context for non-local goto.

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

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

Parameters

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

Returns

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

21.8 <stdint.h>: Standard Integer Types

Exact-width integer types

Integer types having exactly the specified width

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

Integer types capable of holding object pointers

These allow you to declare variables of the same size as a pointer.

- typedef int16_t intptr_t
- typedef uint16_t uintptr_t

Minimum-width integer types

Integer types having at least the specified width

- typedef int8_t int_least8_t
- typedef uint8_t uint_least8_t
- typedef int16_t int_least16_t
- typedef uint16_t uint_least16_t
- typedef int32_t int_least32_t
- typedef uint32_t uint_least32_t
- typedef int64_t int_least64_t
- typedef uint64_t uint_least64_t

Fastest minimum-width integer types

Integer types being usually fastest having at least the specified width

- typedef int8_t int_fast8_t
- typedef uint8_t uint_fast8_t
- typedef int16_t int_fast16_t
- typedef uint16_t uint_fast16_t
- typedef int32_t int_fast32_t
- typedef uint32_t uint_fast32_t
- typedef int64_t int_fast64_t
- typedef uint64_t uint_fast64_t

Greatest-width integer types

Types designating integer data capable of representing any value of any integer type in the corresponding signed or unsigned category

- typedef int64_t intmax_t
- typedef uint64_t uintmax_t

Limits of specified-width integer types

C++ implementations should define these macros only when __STDC_LIMIT_MACROS is defined before <stdint.h> is included

- #define INT8_MAX 0x7f
- #define INT8_MIN (-INT8_MAX - 1)
- #define UINT8_MAX (INT8_MAX * 2 + 1)
- #define INT16_MAX 0x7fff
- #define INT16_MIN (-INT16_MAX - 1)
- #define UINT16_MAX (__CONCAT(INT16_MAX, U) * 2U + 1U)
- #define INT32_MAX 0x7fffffffL
- #define INT32_MIN (-INT32_MAX - 1L)
- #define UINT32_MAX (__CONCAT(INT32_MAX, U) * 2UL + 1UL)
- #define INT64_MAX 0x7fffffffffffffffLL
- #define INT64_MIN (-INT64_MAX - 1LL)
- #define UINT64_MAX (__CONCAT(INT64_MAX, U) * 2ULL + 1ULL)

Limits of minimum-width integer types

- #define INT_LEAST8_MAX INT8_MAX
- #define INT_LEAST8_MIN INT8_MIN
- #define UINT_LEAST8_MAX UINT8_MAX
- #define INT_LEAST16_MAX INT16_MAX
- #define INT_LEAST16_MIN INT16_MIN
- #define UINT_LEAST16_MAX UINT16_MAX
- #define INT_LEAST32_MAX INT32_MAX
- #define INT_LEAST32_MIN INT32_MIN
- #define UINT_LEAST32_MAX UINT32_MAX
- #define INT_LEAST64_MAX INT64_MAX
- #define INT_LEAST64_MIN INT64_MIN
- #define UINT_LEAST64_MAX UINT64_MAX

Limits of fastest minimum-width integer types

- #define INT_FAST8_MAX INT8_MAX
- #define INT_FAST8_MIN INT8_MIN
- #define UINT_FAST8_MAX UINT8_MAX
- #define INT_FAST16_MAX INT16_MAX
- #define INT_FAST16_MIN INT16_MIN
- #define UINT_FAST16_MAX UINT16_MAX
- #define INT_FAST32_MAX INT32_MAX
- #define INT_FAST32_MIN INT32_MIN
- #define UINT_FAST32_MAX UINT32_MAX
- #define INT_FAST64_MAX INT64_MAX
- #define INT_FAST64_MIN INT64_MIN
- #define UINT_FAST64_MAX UINT64_MAX

Limits of integer types capable of holding object pointers

- #define INTPTR_MAX INT16_MAX
- #define INTPTR_MIN INT16_MIN
- #define UINTPTR_MAX UINT16_MAX

Limits of greatest-width integer types

- #define INTMAX_MAX INT64_MAX
- #define INTMAX_MIN INT64_MIN
- #define UINTMAX_MAX UINT64_MAX

Limits of other integer types

C++ implementations should define these macros only when __STDC_LIMIT_MACROS is defined before <stdint.h> is included

- #define PTRDIFF_MAX INT16_MAX
- #define PTRDIFF_MIN INT16_MIN
- #define SIG_ATOMIC_MAX INT8_MAX
- #define SIG_ATOMIC_MIN INT8_MIN
- #define SIZE_MAX UINT16_MAX
- #define **WCHAR_MAX** __WCHAR_MAX__
- #define **WCHAR_MIN** __WCHAR_MIN__
- #define **WINT_MAX** __WINT_MAX__
- #define **WINT_MIN** __WINT_MIN__

Macros for integer constants

C++ implementations should define these macros only when __STDC_CONSTANT_MACROS is defined before <stdint.h> is included.

These definitions are valid for integer constants without suffix and for macros defined as integer constant without suffix

- #define INT8_C(value) ((int8_t) value)
- #define UINT8_C(value) ((uint8_t) __CONCAT(value, U))
- #define INT16_C(value) value
- #define UINT16_C(value) __CONCAT(value, U)
- #define INT32_C(value) __CONCAT(value, L)
- #define UINT32_C(value) __CONCAT(value, UL)
- #define INT64_C(value) __CONCAT(value, LL)
- #define UINT64_C(value) __CONCAT(value, ULL)
- #define INTMAX_C(value) __CONCAT(value, LL)
- #define UINTMAX_C(value) __CONCAT(value, ULL)

21.8.1 Detailed Description

```
#include <stdint.h>
```

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

Since these typedefs are mandated by the C99 standard, they are preferred over rolling your own typedefs.

21.8.2 Macro Definition Documentation

21.8.2.1 INT16_C

```
#define INT16_C(
    value ) value
```

define a constant of type int16_t

21.8.2.2 INT16_MAX

```
#define INT16_MAX 0x7fff
```

largest positive value an int16_t can hold.

21.8.2.3 INT16_MIN

```
#define INT16_MIN (-INT16_MAX - 1)
```

smallest negative value an int16_t can hold.

21.8.2.4 INT32_C

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

define a constant of type int32_t

21.8.2.5 INT32_MAX `#define INT32_MAX 0x7fffffffL`

largest positive value an int32_t can hold.

21.8.2.6 INT32_MIN `#define INT32_MIN (-INT32_MAX - 1L)`

smallest negative value an int32_t can hold.

21.8.2.7 INT64_C `#define INT64_C( value ) __CONCAT(value, LL)`

define a constant of type int64_t

21.8.2.8 INT64_MAX `#define INT64_MAX 0x7fffffffffffffffLL`

largest positive value an int64_t can hold.

21.8.2.9 INT64_MIN `#define INT64_MIN (-INT64_MAX - 1LL)`

smallest negative value an int64_t can hold.

21.8.2.10 INT8_C `#define INT8_C( value ) ((int8_t) value)`

define a constant of type int8_t

21.8.2.11 INT8_MAX `#define INT8_MAX 0x7f`

largest positive value an int8_t can hold.

21.8.2.12 INT8_MIN `#define INT8_MIN (-INT8_MAX - 1)`

smallest negative value an int8_t can hold.

21.8.2.13 INT_FAST16_MAX `#define INT_FAST16_MAX INT16_MAX`

largest positive value an int_fast16_t can hold.

21.8.2.14 INT_FAST16_MIN `#define INT_FAST16_MIN INT16_MIN`

smallest negative value an int_fast16_t can hold.

21.8.2.15 INT_FAST32_MAX `#define INT_FAST32_MAX INT32_MAX`

largest positive value an int_fast32_t can hold.

21.8.2.16 INT_FAST32_MIN `#define INT_FAST32_MIN INT32_MIN`

smallest negative value an int_fast32_t can hold.

21.8.2.17 INT_FAST64_MAX `#define INT_FAST64_MAX INT64_MAX`

largest positive value an int_fast64_t can hold.

21.8.2.18 INT_FAST64_MIN `#define INT_FAST64_MIN INT64_MIN`

smallest negative value an int_fast64_t can hold.

21.8.2.19 INT_FAST8_MAX `#define INT_FAST8_MAX INT8_MAX`

largest positive value an int_fast8_t can hold.

21.8.2.20 INT_FAST8_MIN `#define INT_FAST8_MIN INT8_MIN`

smallest negative value an int_fast8_t can hold.

21.8.2.21 INT_LEAST16_MAX `#define INT_LEAST16_MAX INT16_MAX`

largest positive value an int_least16_t can hold.

21.8.2.22 INT_LEAST16_MIN `#define INT_LEAST16_MIN INT16_MIN`

smallest negative value an int_least16_t can hold.

21.8.2.23 INT_LEAST32_MAX `#define INT_LEAST32_MAX INT32_MAX`

largest positive value an int_least32_t can hold.

21.8.2.24 INT_LEAST32_MIN `#define INT_LEAST32_MIN INT32_MIN`

smallest negative value an int_least32_t can hold.

21.8.2.25 INT_LEAST64_MAX `#define INT_LEAST64_MAX INT64_MAX`

largest positive value an int_least64_t can hold.

21.8.2.26 INT_LEAST64_MIN `#define INT_LEAST64_MIN INT64_MIN`

smallest negative value an int_least64_t can hold.

21.8.2.27 INT_LEAST8_MAX `#define INT_LEAST8_MAX INT8_MAX`

largest positive value an int_least8_t can hold.

21.8.2.28 INT_LEAST8_MIN `#define INT_LEAST8_MIN INT8_MIN`

smallest negative value an int_least8_t can hold.

21.8.2.29 INTMAX_C `#define INTMAX_C( value ) __CONCAT(value, LL)`

define a constant of type intmax_t

21.8.2.30 INTMAX_MAX `#define INTMAX_MAX INT64_MAX`

largest positive value an intmax_t can hold.

21.8.2.31 INTMAX_MIN `#define INTMAX_MIN INT64_MIN`

smallest negative value an intmax_t can hold.

21.8.2.32 INTPTR_MAX `#define INTPTR_MAX INT16_MAX`

largest positive value an intptr_t can hold.

21.8.2.33 INTPTR_MIN `#define INTPTR_MIN INT16_MIN`

smallest negative value an intptr_t can hold.

21.8.2.34 PTRDIFF_MAX `#define PTRDIFF_MAX INT16_MAX`

largest positive value a ptrdiff_t can hold.

21.8.2.35 PTRDIFF_MIN `#define PTRDIFF_MIN INT16_MIN`

smallest negative value a ptrdiff_t can hold.

21.8.2.36 SIG_ATOMIC_MAX `#define SIG_ATOMIC_MAX INT8_MAX`

largest positive value a sig_atomic_t can hold.

21.8.2.37 SIG_ATOMIC_MIN `#define SIG_ATOMIC_MIN INT8_MIN`

smallest negative value a sig_atomic_t can hold.

21.8.2.38 SIZE_MAX `#define SIZE_MAX UINT16_MAX`

largest value a size_t can hold.

21.8.2.39 UINT16_C `#define UINT16_C( value ) __CONCAT(value, U)`

define a constant of type uint16_t

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

largest value an uint16_t can hold.

21.8.2.41 UINT32_C `#define UINT32_C( value ) __CONCAT(value, UL)`

define a constant of type uint32_t

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

largest value an uint32_t can hold.

21.8.2.43 UINT64_C `#define UINT64_C( value ) __CONCAT(value, ULL)`

define a constant of type uint64_t

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

largest value an uint64_t can hold.

21.8.2.45 UINT8_C `#define UINT8_C( value ) ((uint8_t) __CONCAT(value, U))`

define a constant of type uint8_t

21.8.2.46 UINT8_MAX `#define UINT8_MAX (INT8_MAX * 2 + 1)`

largest value an uint8_t can hold.

21.8.2.47 UINT_FAST16_MAX `#define UINT_FAST16_MAX UINT16_MAX`

largest value an uint_fast16_t can hold.

21.8.2.48 UINT_FAST32_MAX `#define UINT_FAST32_MAX UINT32_MAX`

largest value an uint_fast32_t can hold.

21.8.2.49 UINT_FAST64_MAX `#define UINT_FAST64_MAX UINT64_MAX`

largest value an uint_fast64_t can hold.

21.8.2.50 UINT_FAST8_MAX `#define UINT_FAST8_MAX UINT8_MAX`

largest value an uint_fast8_t can hold.

21.8.2.51 UINT_LEAST16_MAX `#define UINT_LEAST16_MAX UINT16_MAX`

largest value an uint_least16_t can hold.

21.8.2.52 UINT_LEAST32_MAX `#define UINT_LEAST32_MAX UINT32_MAX`

largest value an uint_least32_t can hold.

21.8.2.53 UINT_LEAST64_MAX `#define UINT_LEAST64_MAX UINT64_MAX`

largest value an uint_least64_t can hold.

21.8.2.54 UINT_LEAST8_MAX `#define UINT_LEAST8_MAX UINT8_MAX`

largest value an uint_least8_t can hold.

21.8.2.55 UINTMAX_C

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

define a constant of type uintmax_t

21.8.2.56 UINTMAX_MAX `#define UINTMAX_MAX UINT64_MAX`

largest value an uintmax_t can hold.

21.8.2.57 UINTPTR_MAX `#define UINTPTR_MAX UINT16_MAX`

largest value an uintptr_t can hold.

21.8.3 Typedef Documentation

21.8.3.1 int16_t `typedef signed int int16_t`

16-bit signed type.

21.8.3.2 int32_t `typedef signed long int int32_t`

32-bit signed type.

21.8.3.3 int64_t `typedef signed long long int int64_t`

64-bit signed type.

Note

This type is not available when the compiler option -mint8 is in effect.

21.8.3.4 int8_t `typedef signed char int8_t`

8-bit signed type.

21.8.3.5 int_fast16_t `typedef int16_t int_fast16_t`

fastest signed int with at least 16 bits.

21.8.3.6 int_fast32_t `typedef int32_t int_fast32_t`

fastest signed int with at least 32 bits.

21.8.3.7 int_fast64_t `typedef int64_t int_fast64_t`

fastest signed int with at least 64 bits.

Note

This type is not available when the compiler option -mint8 is in effect.

21.8.3.8 int_fast8_t `typedef int8_t int_fast8_t`

fastest signed int with at least 8 bits.

21.8.3.9 int_least16_t `typedef int16_t int_least16_t`

signed int with at least 16 bits.

21.8.3.10 int_least32_t `typedef int32_t int_least32_t`

signed int with at least 32 bits.

21.8.3.11 int_least64_t `typedef int64_t int_least64_t`

signed int with at least 64 bits.

Note

This type is not available when the compiler option -mint8 is in effect.

21.8.3.12 int_least8_t `typedef int8_t int_least8_t`

signed int with at least 8 bits.

21.8.3.13 intmax_t `typedef int64_t intmax_t`

largest signed int available.

21.8.3.14 intptr_t `typedef int16_t intptr_t`

Signed pointer compatible type.

21.8.3.15 uint16_t `typedef unsigned int uint16_t`

16-bit unsigned type.

21.8.3.16 uint32_t `typedef unsigned long int uint32_t`

32-bit unsigned type.

21.8.3.17 uint64_t `typedef unsigned long long int uint64_t`

64-bit unsigned type.

Note

This type is not available when the compiler option -mint8 is in effect.

21.8.3.18 uint8_t `typedef unsigned char uint8_t`

8-bit unsigned type.

21.8.3.19 uint_fast16_t `typedef uint16_t uint_fast16_t`

fastest unsigned int with at least 16 bits.

21.8.3.20 uint_fast32_t `typedef uint32_t uint_fast32_t`

fastest unsigned int with at least 32 bits.

21.8.3.21 uint_fast64_t `typedef uint64_t uint_fast64_t`

fastest unsigned int with at least 64 bits.

Note

This type is not available when the compiler option -mint8 is in effect.

21.8.3.22 uint_fast8_t `typedef uint8_t uint_fast8_t`

fastest unsigned int with at least 8 bits.

21.8.3.23 uint_least16_t `typedef uint16_t uint_least16_t`

unsigned int with at least 16 bits.

21.8.3.24 uint_least32_t `typedef uint32_t uint_least32_t`

unsigned int with at least 32 bits.

21.8.3.25 uint_least64_t `typedef uint64_t uint_least64_t`

unsigned int with at least 64 bits.

Note

This type is not available when the compiler option -mint8 is in effect.

21.8.3.26 uint_least8_t `typedef uint8_t uint_least8_t`

unsigned int with at least 8 bits.

21.8.3.27 uintmax_t `typedef uint64_t uintmax_t`

largest unsigned int available.

21.8.3.28 uintptr_t `typedef uint16_t uintptr_t`

Unsigned pointer compatible type.

21.9 <stdio.h>: Standard IO facilities

Macros

- #define stdin (__iob[0])
- #define stdout (__iob[1])
- #define stderr (__iob[2])
- #define EOF (-1)
- #define fdev_set_udata(stream, u) do { (stream)->udata = u; } while(0)
- #define fdev_get_udata(stream) ((stream)->udata)
- #define fdev_setup_stream(stream, put, get, rwflag)
- #define _FDEV_SETUP_READ __SRD
- #define _FDEV_SETUP_WRITE __SWR
- #define _FDEV_SETUP_RW (__SRD|__SWR)
- #define _FDEV_ERR (-1)
- #define _FDEV_EOF (-2)
- #define FDEV_SETUP_STREAM(put, get, rwflag)
- #define fdev_close()
- #define putc(__c, __stream) fputc(__c, __stream)
- #define putchar(__c) fputc(__c, stdout)
- #define getc(__stream) fgetc(__stream)
- #define getchar() fgetc(stdin)

Typedefs

- typedef struct __file FILE

Functions

- int fclose (FILE *__stream)
- int vfprintf (FILE *__stream, const char *__fmt, va_list __ap)
- int vfprintf_P (FILE *__stream, const char *__fmt, va_list __ap)
- int fputc (int __c, FILE *__stream)
- int printf (const char *__fmt,...)
- int printf_P (const char *__fmt,...)
- int vprintf (const char *__fmt, va_list __ap)
- int sprintf (char *__s, const char *__fmt,...)
- int sprintf_P (char *__s, const char *__fmt,...)
- int snprintf (char *__s, size_t __n, const char *__fmt,...)
- int snprintf_P (char *__s, size_t __n, const char *__fmt,...)
- int vsprintf (char *__s, const char *__fmt, va_list ap)
- int vsprintf_P (char *__s, const char *__fmt, va_list ap)
- int vsnprintf (char *__s, size_t __n, const char *__fmt, va_list ap)
- int vsnprintf_P (char *__s, size_t __n, const char *__fmt, va_list ap)
- int fprintf (FILE *__stream, const char *__fmt,...)
- int fprintf_P (FILE *__stream, const char *__fmt,...)
- int fputs (const char *__str, FILE *__stream)
- int fputs_P (const char *__str, FILE *__stream)
- int puts (const char *__str)
- int puts_P (const char *__str)
- size_t fwrite (const void *__ptr, size_t __size, size_t __nmemb, FILE *__stream)
- int fgetc (FILE *__stream)
- int ungetc (int __c, FILE *__stream)

- char * fgets (char *__str, int __size, FILE *__stream)
- char * gets (char *__str)
- size_t fread (void *__ptr, size_t __size, size_t __nmemb, FILE *__stream)
- void clearerr (FILE *__stream)
- int feof (FILE *__stream)
- int ferror (FILE *__stream)
- int vfscanf (FILE *__stream, const char *__fmt, va_list __ap)
- int vfscanf_P (FILE *__stream, const char *__fmt, va_list __ap)
- int fscanf (FILE *__stream, const char *__fmt,...)
- int fscanf_P (FILE *__stream, const char *__fmt,...)
- int scanf (const char *__fmt,...)
- int scanf_P (const char *__fmt,...)
- int vscanf (const char *__fmt, va_list __ap)
- int sscanf (const char *__buf, const char *__fmt,...)
- int sscanf_P (const char *__buf, const char *__fmt,...)
- int fflush (FILE *stream)
- FILE * fdevopen (int(*put)(char, FILE *), int(*get)(FILE *))

21.9.1 Detailed Description

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

Introduction to the Standard IO facilities This file declares the standard IO facilities that are implemented in AVR-LibC. Due to the nature of the underlying hardware, only a limited subset of standard IO is implemented. There is no actual file implementation available, so only device IO can be performed. Since there's no operating system, the application needs to provide enough details about their devices in order to make them usable by the standard IO facilities.

Due to space constraints, some functionality has not been implemented at all (like some of the `printf` conversions that have been left out). Nevertheless, potential users of this implementation should be warned: the `printf` and `scanf` families of functions, although usually associated with presumably simple things like the famous "Hello, world!" program, are actually fairly complex which causes their inclusion to eat up a fair amount of code space. Also, they are not fast due to the nature of interpreting the format string at run-time. Whenever possible, resorting to the (sometimes non-standard) predetermined conversion facilities that are offered by AVR-LibC will usually cost much less in terms of speed and code size.

Tunable options for code size vs. feature set In order to allow programmers a code size vs. functionality tradeoff, the function vfprintf() which is the heart of the printf family can be selected in different flavours using linker options. See the documentation of vfprintf() for a detailed description. The same applies to vfscanf() and the `scanf` family of functions.

Outline of the chosen API The standard streams `stdin`, `stdout`, and `stderr` are provided, but contrary to the C standard, since AVR-LibC has no knowledge about applicable devices, these streams are not already pre-initialized at application startup. Also, since there is no notion of "file" whatsoever to AVR-LibC, there is no function `fopen()` that could be used to associate a stream to some device. (See note 1.) Instead, the function `fdevopen()` is provided to associate a stream to a device, where the device needs to provide a function to send a character, to receive a character, or both. There is no differentiation between "text" and "binary" streams inside AVR-LibC. Character `\n` is sent literally down to the device's `put()` function. If the device requires a carriage return (`\r`) character to be sent before the linefeed, its `put()` routine must implement this (see note 2).

As an alternative method to fdevopen(), the macro fdev_setup_stream() might be used to setup a user-supplied FILE structure.

It should be noted that the automatic conversion of a newline character into a carriage return - newline sequence breaks binary transfers. If binary transfers are desired, no automatic conversion should be performed, but instead any string that aims to issue a CR-LF sequence must use `"\r\n"` explicitly.

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

It is possible to tie additional user data to a stream, using fdev_set_udata(). The backend put and get functions can then extract this user data using fdev_get_udata(), and act appropriately. For example, a single put function could be used to talk to two different UARTs that way, or the put and get functions could keep internal state between calls there.

Format strings in flash ROM All the `printf` and `scanf` family functions come in two flavours: the standard name, where the format string is expected to be in SRAM, as well as a version with the suffix "_P" where the format string is expected to reside in the flash ROM. The macro PSTR (explained in <avr/pgmspace.h>: Program Space Utilities) becomes very handy for declaring these format strings.

Running stdio without malloc() By default, fdevopen() requires malloc(). As this is often not desired in the limited environment of a microcontroller, an alternative option is provided to run completely without malloc().

The macro fdev_setup_stream() is provided to prepare a user-supplied FILE buffer for operation with stdio.

Example

```
#include <stdio.h>

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

static FILE mystdout = FDEV_SETUP_STREAM(uart_putchar, NULL,
                                         _FDEV_SETUP_WRITE);

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

  if (c == '\n')
    uart_putchar('\r', stream);
  loop_until_bit_is_set(UCSRA, UDRE);
  UDR = c;
  return 0;
}

int
main(void)
{
  init_uart();
  stdout = &mystdout;
  printf("Hello, world!\n");

  return 0;
}
```

This example uses the initializer form FDEV_SETUP_STREAM() rather than the function-like fdev_setup_stream(), so all data initialization happens during C start-up.

If streams initialized that way are no longer needed, they can be destroyed by first calling the macro fdev_close(), and then destroying the object itself. No call to fclose() should be issued for these streams. While calling fclose() itself is harmless, it will cause an undefined reference to free() and thus cause the linker to link the malloc module into the application.

Notes

Note 1: It might have been possible to implement a device abstraction that is compatible with `fopen()` but since this would have required to parse a string, and to take all the information needed either out of this string, or out of an additional table that would need to be provided by the application, this approach was not taken.

Note 2: This basically follows the Unix approach: if a device such as a terminal needs special handling, it is in the domain of the terminal device driver to provide this functionality. Thus, a simple function suitable as `put()` for `fdevopen()` that talks to a UART interface might look like this:

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

  if (c == '\n')
    uart_putchar('\r', stream);
  loop_until_bit_is_set(UCSRA, UDRE);
  UDR = c;
  return 0;
}
```

Note 3: This implementation has been chosen because the cost of maintaining an alias is considerably smaller than the cost of maintaining full copies of each stream. Yet, providing an implementation that offers the complete set of standard streams was deemed to be useful. Not only that writing `printf()` instead of `fprintf(mystream, ...)` saves typing work, but since avr-gcc needs to resort to pass all arguments of variadic functions on the stack (as opposed to passing them in registers for functions that take a fixed number of parameters), the ability to pass one parameter less by implying `stdin` or stdout will also save some execution time.

21.9.2 Macro Definition Documentation

21.9.2.1 _FDEV_EOF `#define _FDEV_EOF (-2)`

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

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

21.9.2.2 _FDEV_ERR `#define _FDEV_ERR (-1)`

Return code for an error condition during device read.

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

21.9.2.3 _FDEV_SETUP_READ `#define _FDEV_SETUP_READ __SRD`

fdev_setup_stream() with read intent

21.9.2.4 _FDEV_SETUP_RW `#define _FDEV_SETUP_RW (__SRD|__SWR)`

fdev_setup_stream() with read/write intent

21.9.2.5 _FDEV_SETUP_WRITE `#define _FDEV_SETUP_WRITE __SWR`

fdev_setup_stream() with write intent

21.9.2.6 EOF `#define EOF (-1)`

`EOF` declares the value that is returned by various standard IO functions in case of an error. Since the AVR platform (currently) doesn't contain an abstraction for actual files, its origin as "end of file" is somewhat meaningless here.

21.9.2.7 fdev_close `#define fdev_close( )`

This macro frees up any library resources that might be associated with `stream`. It should be called if `stream` is no longer needed, right before the application is going to destroy the `stream` object itself.

(Currently, this macro evaluates to nothing, but this might change in future versions of the library.)

21.9.2.8 fdev_get_udata

```
#define fdev_get_udata(
        stream ) ((stream)->udata)
```

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

21.9.2.9 fdev_set_udata

```
#define fdev_set_udata(
        stream,
        u ) do { (stream)->udata = u; } while(0)
```

This macro inserts a pointer to user defined data into a FILE stream object.

The user data can be useful for tracking state in the put and get functions supplied to the fdevopen() function.

21.9.2.10 FDEV_SETUP_STREAM

```
#define FDEV_SETUP_STREAM(
        put,
        get,
        rwflag )
```

Initializer for a user-supplied stdio stream.

This macro acts similar to fdev_setup_stream(), but it is to be used as the initializer of a variable of type FILE.

The remaining arguments are to be used as explained in fdev_setup_stream().

21.9.2.11 fdev_setup_stream

```
#define fdev_setup_stream(
        stream,
        put,
        get,
        rwflag )
```

Setup a user-supplied buffer as an stdio stream.

This macro takes a user-supplied buffer `stream`, and sets it up as a stream that is valid for stdio operations, similar to one that has been obtained dynamically from fdevopen(). The buffer to setup must be of type FILE.

The arguments `put` and `get` are identical to those that need to be passed to fdevopen().

The `rwflag` argument can take one of the values _FDEV_SETUP_READ, _FDEV_SETUP_WRITE, or _FDEV_SETUP_RW, for read, write, or read/write intent, respectively.

Note

> No assignments to the standard streams will be performed by fdev_setup_stream(). If standard streams are to be used, these need to be assigned by the user. See also under Running stdio without malloc().

21.9.2.12 getc

```
#define getc(
    __stream ) fgetc(__stream)
```

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

21.9.2.13 getchar

```
#define getchar(
    void ) fgetc(stdin)
```

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

21.9.2.14 putc

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

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

21.9.2.15 putchar

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

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

21.9.2.16 stderr

```
#define stderr (__iob[2])
```

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

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

21.9.2.17 stdin

```
#define stdin (__iob[0])
```

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

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

21.9.2.18 stdout

```
#define stdout (__iob[1])
```

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

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

21.9.3 Typedef Documentation

21.9.3.1 FILE

```
typedef struct __file FILE
```

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

21.9.4 Function Documentation

21.9.4.1 clearerr()

```
void clearerr (
    FILE * __stream )
```

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

21.9.4.2 fclose()

```
int fclose (
    FILE * __stream )
```

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

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

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

It currently always returns 0 (for success).

21.9.4.3 fdevopen()

```
FILE * fdevopen (
    int(*)(char, FILE *) put,
    int(*)(FILE *) get )
```

This function is a replacement for `fopen()`.

It opens a stream for a device where the actual device implementation needs to be provided by the application. If successful, a pointer to the structure for the opened stream is returned. Reasons for a possible failure currently include that neither the `put` nor the `get` argument have been provided, thus attempting to open a stream with no IO intent at all, or that insufficient dynamic memory is available to establish a new stream.

If the `put` function pointer is provided, the stream is opened with write intent. The function passed as `put` shall take two arguments, the first a character to write to the device, and the second a pointer to FILE, and shall return 0 if the output was successful, and a nonzero value if the character could not be sent to the device.

If the `get` function pointer is provided, the stream is opened with read intent. The function passed as `get` shall take a pointer to FILE as its single argument, and return one character from the device, passed as an `int` type. If an error occurs when trying to read from the device, it shall return `_FDEV_ERR`. If an end-of-file condition was reached while reading from the device, `_FDEV_EOF` shall be returned.

If both functions are provided, the stream is opened with read and write intent.

The first stream opened with read intent is assigned to `stdin`, and the first one opened with write intent is assigned to both, `stdout` and `stderr`.

fdevopen() uses calloc() (und thus malloc()) in order to allocate the storage for the new stream.

Note

> If the macro __STDIO_FDEVOPEN_COMPAT_12 is declared before including <stdio.h>, a function prototype for fdevopen() will be chosen that is backwards compatible with AVR-LibC version 1.2 and before. This is solely intented for providing a simple migration path without the need to immediately change all source code. Do not use for new code.

21.9.4.4 feof()

```
int feof (
    FILE * __stream )
```

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

21.9.4.5 ferror()

```
int ferror (
    FILE * __stream )
```

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

21.9.4.6 fflush()

```
int fflush (
    FILE * stream )
```

Flush `stream`.

This is a null operation provided for source-code compatibility only, as the standard IO implementation currently does not perform any buffering.

21.9.4.7 fgetc()

```
int fgetc (
    FILE * __stream )
```

The function `fgetc` reads a character from `stream`. It returns the character, or `EOF` in case end-of-file was encountered or an error occurred. The routines feof() or ferror() must be used to distinguish between both situations.

21.9.4.8 fgets()

```
char * fgets (
    char * __str,
    int __size,
    FILE * __stream )
```

Read at most `size - 1` bytes from `stream`, until a newline character was encountered, and store the characters in the buffer pointed to by `str`. Unless an error was encountered while reading, the string will then be terminated with a `NUL` character.

If an error was encountered, the function returns NULL and sets the error flag of `stream`, which can be tested using ferror(). Otherwise, a pointer to the string will be returned.

21.9.4.9 fprintf()

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

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

21.9.4.10 fprintf_P()

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

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

21.9.4.11 fputc()

```
int fputc (
    int __c,
    FILE * __stream )
```

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

21.9.4.12 fputs()

```
int fputs (
    const char * __str,
    FILE * __stream )
```

Write the string pointed to by `str` to stream `stream`.

Returns 0 on success and EOF on error.

21.9.4.13 fputs_P()

```
int fputs_P (
    const char * __str,
    FILE * __stream )
```

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

21.9.4.14 fread()

```
size_t fread (
    void * __ptr,
    size_t __size,
    size_t __nmemb,
    FILE * __stream )
```

Read `nmemb` objects, `size` bytes each, from `stream`, to the buffer pointed to by `ptr`.

Returns the number of objects successfully read, i. e. `nmemb` unless an input error occured or end-of-file was encountered. feof() and ferror() must be used to distinguish between these two conditions.

21.9.4.15 fscanf()

```
int fscanf (
    FILE * __stream,
    const char * __fmt,
     ... )
```

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

See vfscanf() for details.

21.9.4.16 fscanf_P()

```
int fscanf_P (
    FILE * __stream,
    const char * __fmt,
     ... )
```

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

21.9.4.17 fwrite()

```
size_t fwrite (
    const void * __ptr,
    size_t __size,
    size_t __nmemb,
    FILE * __stream )
```

Write `nmemb` objects, `size` bytes each, to `stream`. The first byte of the first object is referenced by `ptr`.

Returns the number of objects successfully written, i. e. `nmemb` unless an output error occured.

21.9.4.18 gets()

```
char * gets (
    char * __str )
```

Similar to fgets() except that it will operate on stream `stdin`, and the trailing newline (if any) will not be stored in the string. It is the caller's responsibility to provide enough storage to hold the characters read.

21.9.4.19 printf()

```
int printf (
    const char * __fmt,
    ... )
```

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

21.9.4.20 printf_P()

```
int printf_P (
    const char * __fmt,
    ... )
```

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

21.9.4.21 puts()

```
int puts (
    const char * __str )
```

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

21.9.4.22 puts_P()

```
int puts_P (
    const char * __str )
```

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

21.9.4.23 scanf()

```
int scanf (
    const char * __fmt,
    ... )
```

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

See vfscanf() for details.

21.9.4.24 scanf_P()

```
int scanf_P (
    const char * __fmt,
    ... )
```

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

21.9.4.25 snprintf()

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

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

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

21.9.4.26 snprintf_P()

```
int snprintf_P (
    char * __s,
    size_t __n,
    const char * __fmt,
     ... )
```

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

21.9.4.27 sprintf()

```
int sprintf (
    char * __s,
    const char * __fmt,
     ... )
```

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

21.9.4.28 sprintf_P()

```
int sprintf_P (
    char * __s,
    const char * __fmt,
     ... )
```

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

21.9.4.29 sscanf()

```
int sscanf (
    const char * __buf,
    const char * __fmt,
     ... )
```

The function `sscanf` performs formatted input, reading the input data from the buffer pointed to by `buf`.

See vfscanf() for details.

21.9.4.30 sscanf_P()

```
int sscanf_P (
    const char * __buf,
    const char * __fmt,
     ... )
```

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

21.9.4.31 ungetc()

```
int ungetc (
    int __c,
    FILE * __stream )
```

The ungetc() function pushes the character c (converted to an unsigned char) back onto the input stream pointed to by stream. The pushed-back character will be returned by a subsequent read on the stream.

Currently, only a single character can be pushed back onto the stream.

The ungetc() function returns the character pushed back after the conversion, or EOF if the operation fails. If the value of the argument c character equals EOF, the operation will fail and the stream will remain unchanged.

21.9.4.32 vfprintf()

```
int vfprintf (
    FILE * __stream,
    const char * __fmt,
    va_list __ap )
```

vfprintf is the central facility of the printf family of functions. It outputs values to stream under control of a format string passed in fmt. The actual values to print are passed as a variable argument list ap.

vfprintf returns the number of characters written to stream, or EOF in case of an error. Currently, this will only happen if stream has not been opened with write intent.

The format string is composed of zero or more directives: ordinary characters (not %), which are copied unchanged to the output stream; and conversion specifications, each of which results in fetching zero or more subsequent arguments. Each conversion specification is introduced by the % character. The arguments must properly correspond (after type promotion) with the conversion specifier. After the %, the following appear in sequence:

- Zero or more of the following flags:
  - # The value should be converted to an "alternate form". For c, d, i, s, and u conversions, this option has no effect. For o conversions, the precision of the number is increased to force the first character of the output string to a zero (except if a zero value is printed with an explicit precision of zero). For x and X conversions, a non-zero result has the string `0x' (or `0X' for X conversions) prepended to it.
  - 0 (zero) Zero padding. For all conversions, the converted value is padded on the left with zeros rather than blanks. If a precision is given with a numeric conversion (d, i, o, u, i, x, and X), the 0 flag is ignored.
  - \- A negative field width flag; the converted value is to be left adjusted on the field boundary. The converted value is padded on the right with blanks, rather than on the left with blanks or zeros. A - overrides a 0 if both are given.
  - ' ' (space) A blank should be left before a positive number produced by a signed conversion (d, or i).
  - \+ A sign must always be placed before a number produced by a signed conversion. A + overrides a space if both are used.
- An optional decimal digit string specifying a minimum field width. If the converted value has fewer characters than the field width, it will be padded with spaces on the left (or right, if the left-adjustment flag has been given) to fill out the field width.
- An optional precision, in the form of a period . followed by an optional digit string. If the digit string is omitted, the precision is taken as zero. This gives the minimum number of digits to appear for d, i, o, u, x, and X conversions, or the maximum number of characters to be printed from a string for s conversions.
- An optional l or h length modifier, that specifies that the argument for the d, i, o, u, x, or X conversion is a "long int" rather than int. The h is ignored, as "short int" is equivalent to int.
- A character that specifies the type of conversion to be applied.

The conversion specifiers and their meanings are:

- `diouxX` The int (or appropriate variant) argument is converted to signed decimal (d and i), unsigned octal (o), unsigned decimal (u), or unsigned hexadecimal (x and X) notation. The letters "abcdef" are used for x conversions; the letters "ABCDEF" are used for X conversions. The precision, if any, gives the minimum number of digits that must appear; if the converted value requires fewer digits, it is padded on the left with zeros.
- `p` The `void *` argument is taken as an unsigned integer, and converted similarly as a `%#x` command would do.
- `c` The `int` argument is converted to an `"unsigned char"`, and the resulting character is written.
- `s` The `"char *"` argument is expected to be a pointer to an array of character type (pointer to a string). Characters from the array are written up to (but not including) a terminating NUL character; if a precision is specified, no more than the number specified are written. If a precision is given, no null character need be present; if the precision is not specified, or is greater than the size of the array, the array must contain a terminating NUL character.
- `%` A `%` is written. No argument is converted. The complete conversion specification is "%%".
- `eE` The double argument is rounded and converted in the format `"[-]d.ddde±dd"` where there is one digit before the decimal-point character and the number of digits after it is equal to the precision; if the precision is missing, it is taken as 6; if the precision is zero, no decimal-point character appears. An *E* conversion uses the letter `'E'` (rather than `'e'`) to introduce the exponent. The exponent always contains two digits; if the value is zero, the exponent is 00.
- `fF` The double argument is rounded and converted to decimal notation in the format `"[-]ddd.ddd"`, where the number of digits after the decimal-point character is equal to the precision specification. If the precision is missing, it is taken as 6; if the precision is explicitly zero, no decimal-point character appears. If a decimal point appears, at least one digit appears before it.
- `gG` The double argument is converted in style `f` or `e` (or `F` or `E` for `G` conversions). The precision specifies the number of significant digits. If the precision is missing, 6 digits are given; if the precision is zero, it is treated as 1. Style `e` is used if the exponent from its conversion is less than -4 or greater than or equal to the precision. Trailing zeros are removed from the fractional part of the result; a decimal point appears only if it is followed by at least one digit.
- `S` Similar to the `s` format, except the pointer is expected to point to a program-memory (ROM) string instead of a RAM string.

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

Since the full implementation of all the mentioned features becomes fairly large, three different flavours of vfprintf() can be selected using linker options. The default vfprintf() implements all the mentioned functionality except floating point conversions. A minimized version of vfprintf() is available that only implements the very basic integer and string conversion facilities, but only the `#` additional option can be specified using conversion flags (these flags are parsed correctly from the format specification, but then simply ignored). This version can be requested using the following compiler options:

```
-Wl,-u,vfprintf -lprintf_min
```

If the full functionality including the floating point conversions is required, the following options should be used:

```
-Wl,-u,vfprintf -lprintf_flt -lm
```

Limitations:

- The specified width and precision can be at most 255.

Notes:

- For floating-point conversions, if you link default or minimized version of vfprintf(), the symbol ? will be output and double argument will be skipped. So you output below will not be crashed. For default version the width field and the "pad to left" (symbol minus) option will work in this case.
- The `hh` length modifier is ignored (`char` argument is promouted to `int`). More exactly, this realization does not check the number of `h` symbols.
- But the `ll` length modifier will to abort the output, as this realization does not operate `long long` arguments.
- The variable width or precision field (an asterisk * symbol) is not realized and will to abort the output.

21.9.4.33 vfprintf_P()

```
int vfprintf_P (
    FILE * __stream,
    const char * __fmt,
    va_list __ap )
```

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

21.9.4.34 vfscanf()

```
int vfscanf (
    FILE * stream,
    const char * fmt,
    va_list ap )
```

Formatted input. This function is the heart of the **scanf** family of functions.

Characters are read from *stream* and processed in a way described by *fmt*. Conversion results will be assigned to the parameters passed via *ap*.

The format string *fmt* is scanned for conversion specifications. Anything that doesn't comprise a conversion specification is taken as text that is matched literally against the input. White space in the format string will match any white space in the data (including none), all other characters match only itself. Processing is aborted as soon as the data and format string no longer match, or there is an error or end-of-file condition on *stream*.

Most conversions skip leading white space before starting the actual conversion.

Conversions are introduced with the character **%**. Possible options can follow the **%**:

- a * indicating that the conversion should be performed but the conversion result is to be discarded; no parameters will be processed from `ap`,
- the character `h` indicating that the argument is a pointer to `short int` (rather than `int`),
- the 2 characters `hh` indicating that the argument is a pointer to `char` (rather than `int`).
- the character `l` indicating that the argument is a pointer to `long int` (rather than `int`, for integer type conversions), or a pointer to `float` (for floating point conversions),

In addition, a maximal field width may be specified as a nonzero positive decimal integer, which will restrict the conversion to at most this many characters from the input stream. This field width is limited to at most 255 characters which is also the default value (except for the `c` conversion that defaults to 1).

The following conversion flags are supported:

- % Matches a literal % character. This is not a conversion.
- d Matches an optionally signed decimal integer; the next pointer must be a pointer to int.
- i Matches an optionally signed integer; the next pointer must be a pointer to int. The integer is read in base 16 if it begins with **0x** or **0X**, in base 8 if it begins with **0**, and in base 10 otherwise. Only characters that correspond to the base are used.
- o Matches an octal integer; the next pointer must be a pointer to unsigned int.
- u Matches an optionally signed decimal integer; the next pointer must be a pointer to unsigned int.
- x Matches an optionally signed hexadecimal integer; the next pointer must be a pointer to unsigned int.
- f Matches an optionally signed floating-point number; the next pointer must be a pointer to float.
- e, g, F, E, G Equivalent to f.
- s Matches a sequence of non-white-space characters; the next pointer must be a pointer to char, and the array must be large enough to accept all the sequence and the terminating NUL character. The input string stops at white space or at the maximum field width, whichever occurs first.
- c Matches a sequence of width count characters (default 1); the next pointer must be a pointer to char, and there must be enough room for all the characters (no terminating NUL is added). The usual skip of leading white space is suppressed. To skip white space first, use an explicit space in the format.
- [Matches a nonempty sequence of characters from the specified set of accepted characters; the next pointer must be a pointer to char, and there must be enough room for all the characters in the string, plus a terminating NUL character. The usual skip of leading white space is suppressed. The string is to be made up of characters in (or not in) a particular set; the set is defined by the characters between the open bracket [character and a close bracket] character. The set excludes those characters if the first character after the open bracket is a circumflex ^. To include a close bracket in the set, make it the first character after the open bracket or the circumflex; any other position will end the set. The hyphen character - is also special; when placed between two other characters, it adds all intervening characters to the set. To include a hyphen, make it the last character before the final close bracket. For instance, [^]0-9-] means the set of *everything except close bracket, zero through nine, and hyphen*. The string ends with the appearance of a character not in the (or, with a circumflex, in) set or when the field width runs out. Note that usage of this conversion enlarges the stack expense.
- p Matches a pointer value (as printed by p in printf()); the next pointer must be a pointer to void.
- n Nothing is expected; instead, the number of characters consumed thus far from the input is stored through the next pointer, which must be a pointer to int. This is not a conversion, although it can be suppressed with the * flag.

  These functions return the number of input items assigned, which can be fewer than provided for, or even zero, in the event of a matching failure. Zero indicates that, while there was input available, no conversions were assigned; typically this is due to an invalid input character, such as an alphabetic character for a d conversion. The value EOF is returned if an input failure occurs before any conversion such as an end-of-file occurs. If an error or end-of-file occurs after conversion has begun, the number of conversions which were successfully completed is returned.

  By default, all the conversions described above are available except the floating-point conversions and the width is limited to 255 characters. The float-point conversion will be available in the extended version provided by the library libscanf_flt.a. Also in this case the width is not limited (exactly, it is limited to 65535 characters). To link a program against the extended version, use the following compiler flags in the link stage:

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

  A third version is available for environments that are tight on space. In addition to the restrictions of the standard one, this version implements no %[specification. This version is provided in the library libscanf_min.a, and can be requested using the following options in the link stage:

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

21.9.4.35 vfscanf_P() int vfscanf_P (

```
FILE * __stream,
const char * __fmt,
va_list __ap )
```

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

21.9.4.36 vprintf() int vprintf (

```
const char * __fmt,
va_list __ap )
```

The function `vprintf` performs formatted output to stream `stdout`, taking a variable argument list as in vfprintf().

See vfprintf() for details.

21.9.4.37 vscanf() int vscanf (

```
const char * __fmt,
va_list __ap )
```

The function `vscanf` performs formatted input from stream `stdin`, taking a variable argument list as in vfscanf().

See vfscanf() for details.

21.9.4.38 vsnprintf() int vsnprintf (

```
char * __s,
size_t __n,
const char * __fmt,
va_list ap )
```

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

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

21.9.4.39 vsnprintf_P() int vsnprintf_P (

```
char * __s,
size_t __n,
const char * __fmt,
va_list ap )
```

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

21.9.4.40 vsprintf() int vsprintf (

```
char * __s,
const char * __fmt,
va_list ap )
```

Like `sprintf()` but takes a variable argument list for the arguments.

21.9.4.41 vsprintf_P()

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

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

21.10 <stdlib.h>: General utilities

Data Structures

- struct div_t
- struct ldiv_t

Macros

- #define RAND_MAX 0x7FFF

Typedefs

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

Functions

- void abort (void)
- int abs (int __i)
- long labs (long __i)
- void ∗ bsearch (const void ∗__key, const void ∗__base, size_t __nmemb, size_t __size, int(∗__compar)(const void ∗, const void ∗))
- div_t div (int __num, int __denom) __asm__("__divmodhi4")
- ldiv_t ldiv (long __num, long __denom) __asm__("__divmodsi4")
- void qsort (void ∗__base, size_t __nmemb, size_t __size, __compar_fn_t __compar)
- long strtol (const char ∗__nptr, char ∗∗__endptr, int __base)
- unsigned long strtoul (const char ∗__nptr, char ∗∗__endptr, int __base)
- long atol (const char ∗__s)
- int atoi (const char ∗__s)
- void exit (int __status)
- void ∗ malloc (size_t __size)
- void free (void ∗__ptr)
- void ∗ calloc (size_t __nele, size_t __size)
- void ∗ realloc (void ∗__ptr, size_t __size)
- float strtof (const char ∗__nptr, char ∗∗__endptr)
- double strtod (const char ∗__nptr, char ∗∗__endptr)
- long double strtold (const char ∗__nptr, char ∗∗__endptr)
- int atexit (void(∗func)(void))
- float atoff (const char ∗__nptr)
- double atof (const char ∗__nptr)
- long double atofl (const char ∗__nptr)
- int rand (void)
- void srand (unsigned int __seed)
- int rand_r (unsigned long ∗__ctx)

Variables

- size_t __malloc_margin
- char * __malloc_heap_start
- char * __malloc_heap_end

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

- char * ltoa (long val, char *s, int radix)
- char * utoa (unsigned int val, char *s, int radix)
- char * ultoa (unsigned long val, char *s, int radix)
- long random (void)
- void srandom (unsigned long __seed)
- long random_r (unsigned long *__ctx)
- char * itoa (int val, char *s, int radix)
- #define RANDOM_MAX 0x7FFFFFFF

Conversion functions for double arguments.

- char * ftostre (float __val, char *__s, unsigned char __prec, unsigned char __flags)
- char * dtostre (double __val, char *__s, unsigned char __prec, unsigned char __flags)
- char * ldtostre (long double __val, char *__s, unsigned char __prec, unsigned char __flags)
- char * ftostrf (float __val, signed char __width, unsigned char __prec, char *__s)
- char * dtostrf (double __val, signed char __width, unsigned char __prec, char *__s)
- char * ldtostrf (long double __val, signed char __width, unsigned char __prec, char *__s)
- #define DTOSTR_ALWAYS_SIGN 0x01 /* put '+' or ' ' for positives */
- #define DTOSTR_PLUS_SIGN 0x02 /* put '+' rather than ' ' */
- #define DTOSTR_UPPERCASE 0x04 /* put 'E' rather 'e' */
- #define EXIT_SUCCESS 0
- #define EXIT_FAILURE 1

21.10.1 Detailed Description

```
#include <stdlib.h>
```

This file declares some basic C macros and functions as defined by the ISO standard, plus some AVR-specific extensions.

21.10.2 Macro Definition Documentation

21.10.2.1 DTOSTR_ALWAYS_SIGN `#define DTOSTR_ALWAYS_SIGN 0x01 /* put '+' or ' ' for positives */`

Bit value that can be passed in `flags` to ftostre(), dtostre() and ldtostre().

21.10.2.2 DTOSTR_PLUS_SIGN `#define DTOSTR_PLUS_SIGN 0x02 /* put '+' rather than ' ' */`

Bit value that can be passed in `flags` to ftostre(), dtostre() and ldtostre().

21.10.2.3 DTOSTR_UPPERCASE `#define DTOSTR_UPPERCASE 0x04 /* put 'E' rather 'e' */`

Bit value that can be passed in `flags` to ftostre(), dtostre() and ldtostre().

21.10.2.4 EXIT_FAILURE `#define EXIT_FAILURE 1`

Unsuccessful termination for exit(); evaluates to a non-zero value.

21.10.2.5 EXIT_SUCCESS `#define EXIT_SUCCESS 0`

Successful termination for exit(); evaluates to 0.

21.10.2.6 RAND_MAX `#define RAND_MAX 0x7FFF`

Highest number that can be generated by rand().

21.10.2.7 RANDOM_MAX `#define RANDOM_MAX 0x7FFFFFFF`

Highest number that can be generated by random().

21.10.3 Typedef Documentation

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

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

21.10.4 Function Documentation

21.10.4.1 abort()

```
void abort (
    void )
```

The abort() function causes abnormal program termination to occur. This realization disables interrupts and jumps to _exit() function with argument equal to 1. In the limited AVR environment, execution is effectively halted by entering an infinite loop.

21.10.4.2 abs()

```
int abs (
    int __i )
```

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

Note

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

21.10.4.3 atexit() int atexit (

void(*)(void) *func*)

The atexit() function registers function *func* to be run as part of the `exit()` function during .fini8. atexit() calls malloc().

21.10.4.4 atof() double atof (

const char * *nptr*)

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

It is equivalent to calling

```
strtod(nptr, (char**) 0);
```

21.10.4.5 atoff() float atoff (

const char * *nptr*)

The atoff() function converts the initial portion of the string pointed to by *nptr* to `float` representation.

It is equivalent to calling

```
strtof(nptr, (char**) 0);
```

21.10.4.6 atofl() long double atofl (

const char * *nptr*)

The atofl() function converts the initial portion of the string pointed to by *nptr* to `long double` representation.

It is equivalent to calling

```
strtold(nptr, (char**) 0);
```

21.10.4.7 atoi() int atoi (

const char * *__s*)

The atoi() function converts the initial portion of the string pointed to by `s` to integer representation. In contrast to

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

this function does not detect overflow (`errno` is not changed and the result value is not predictable), uses smaller memory (flash and stack) and works more quickly.

21.10.4.8 atol() long atol (

const char * *__s*)

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

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

this function does not detect overflow (`errno` is not changed and the result value is not predictable), uses smaller memory (flash and stack) and works more quickly.

21.10.4.9 bsearch()

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

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

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

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

21.10.4.10 calloc()

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

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

21.10.4.11 div()

```
div_t div (
    int __num,
    int __denom )
```

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

21.10.4.12 dtostre()

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

The dtostre() function is similar to the ftostre() function, except that it converts a `double` value instead of a `float` value.

dtostre() is currently only supported when `double` is a 32-bit type.

21.10.4.13 dtostrf()

```
char * dtostrf (
    double __val,
    signed char __width,
    unsigned char __prec,
    char * __s )
```

The dtostrf() function is similar to the ftostrf() function, except that converts a `double` value instead of a `float` value.

ldtostre() is currently only supported when `double` is a 32-bit type.

21.10.4.14 exit()

```
void exit (
    int __status )
```

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

Global destructors will be called before halting execution, see the .fini sections.

21.10.4.15 free()

```
void free (
    void * __ptr )
```

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

21.10.4.16 ftostre()

```
char * ftostre (
    float __val,
    char * __s,
    unsigned char __prec,
    unsigned char __flags )
```

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

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

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

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

The ftostre() function returns the pointer to the converted string `s`.

21.10.4.17 ftostrf()

```
char * ftostrf (
    float __val,
    signed char __width,
    unsigned char __prec,
    char * __s )
```

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

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

The ftostrf() function returns the pointer to the converted string `s`.

21.10.4.18 itoa()

```
char * itoa (
    int val,
    char * s,
    int radix )
```

Convert an integer to a string.

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

Note

The minimal size of the buffer `s` depends on the choice of radix. For example, if the radix is 2 (binary), you need to supply a buffer with a minimal length of 8 ∗ sizeof (int) + 1 characters, i.e. one character for each bit plus one for the string terminator. Using a larger radix will require a smaller minimal buffer size.

Warning

If the buffer is too small, you risk a buffer overflow.

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

If radix is 10 and val is negative, a minus sign will be prepended.

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

21.10.4.19 labs()

```
long labs (
    long __i )
```

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

Note

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

21.10.4.20 ldiv()

```
ldiv_t ldiv (
    long __num,
    long __denom )
```

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

21.10.4.21 ldtostre()

```
char * ldtostre (
    long double __val,
    char * __s,
    unsigned char __prec,
    unsigned char __flags )
```

The ldtostre() function is similar to the ftostre() function, except that it converts a `long double` value instead of a `float` value.

ldtostre() is currently only supported when `long double` is a 32-bit type.

21.10.4.22 ldtostrf()

```
char * ldtostrf (
    long double __val,
    signed char __width,
    unsigned char __prec,
    char * __s )
```

The ldtostrf() function is similar to the ftostrf() function, except that converts a `long double` value instead of a `float` value.

ldtostre() is currently only supported when `long double` is a 32-bit type.

21.10.4.23 ltoa()

```
char * ltoa (
    long val,
    char * s,
    int radix )
```

Convert a long integer to a string.

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

Note

> The minimal size of the buffer `s` depends on the choice of radix. For example, if the radix is 2 (binary), you need to supply a buffer with a minimal length of 8 ∗ sizeof (long int) + 1 characters, i.e. one character for each bit plus one for the string terminator. Using a larger radix will require a smaller minimal buffer size.

Warning

> If the buffer is too small, you risk a buffer overflow.

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

If radix is 10 and val is negative, a minus sign will be prepended.

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

21.10.4.24 malloc()

```
void * malloc (
    size_t __size )
```

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

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

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

21.10.4.25 qsort()

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

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

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

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

21.10.4.26 rand()

```
int rand (
    void )
```

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

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

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

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

21.10.4.27 rand_r()

```
int rand_r (
    unsigned long * __ctx )
```

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

21.10.4.28 random()

```
long random (
    void )
```

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

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

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

21.10.4.29 random_r()

```
long random_r (
    unsigned long * __ctx )
```

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

21.10.4.30 realloc()

```
void * realloc (
    void * __ptr,
    size_t __size )
```

The realloc() function tries to change the size of the region allocated at `ptr` to the new `size` value. It returns a pointer to the new region. The returned pointer might be the same as the old pointer, or a pointer to a completely different region.

The contents of the returned region up to either the old or the new size value (whatever is less) will be identical to the contents of the old region, even in case a new region had to be allocated.

It is acceptable to pass `ptr` as NULL, in which case realloc() will behave identical to malloc().

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

21.10.4.31 srand()

```
void srand (
    unsigned int __seed )
```

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

21.10.4.32 srandom()

```
void srandom (
    unsigned long __seed )
```

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

21.10.4.33 strtod()

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

The strtod() function is similar to strtof(), except that the conversion result is of type `double` instead of `float`.

strtod() is currently only supported when `double` is a 32-bit type.

21.10.4.34 strtof()

```
float strtof (
    const char * nptr,
    char ** endptr )
```

The strtof() function converts the initial portion of the string pointed to by *nptr* to `float` representation.

The expected form of the string is an optional plus (`'+'`) or minus sign (`'-'`) followed by a sequence of digits optionally containing a decimal-point character, optionally followed by an exponent. An exponent consists of an `'E'` or `'e'`, followed by an optional plus or minus sign, followed by a sequence of digits.

Leading white-space characters in the string are skipped.

The strtof() function returns the converted value, if any.

If *endptr* is not `NULL`, a pointer to the character after the last character used in the conversion is stored in the location referenced by *endptr*.

If no conversion is performed, zero is returned and the value of *nptr* is stored in the location referenced by *endptr*.

If the correct value would cause overflow, plus or minus `INFINITY` is returned (according to the sign of the value), and `ERANGE` is stored in `errno`. If the correct value would cause underflow, zero is returned and `ERANGE` is stored in `errno`.

21.10.4.35 strtol()

```
long strtol (
    const char * __nptr,
    char ** __endptr,
    int __base )
```

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

The string may begin with an arbitrary amount of white space (as determined by isspace()) followed by a single optional `'+'` or `'-'` sign. If `base` is zero or 16, the string may then include a `"0x"` prefix, and the number will be read in base 16; otherwise, a zero base is taken as 10 (decimal) unless the next character is `'0'`, in which case it is taken as 8 (octal).

The remainder of the string is converted to a long value in the obvious manner, stopping at the first character which is not a valid digit in the given base. (In bases above 10, the letter `'A'` in either upper or lower case represents 10, `'B'` represents 11, and so forth, with `'Z'` representing 35.)

If `endptr` is not NULL, strtol() stores the address of the first invalid character in `*endptr`. If there were no digits at all, however, strtol() stores the original value of `nptr` in `endptr`. (Thus, if `*nptr` is not `'\0'` but `**endptr` is `'\0'` on return, the entire string was valid.)

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

21.10.4.36 strtold()

```
long double strtold (
    const char * __nptr,
    char ** __endptr )
```

The strtold() function is similar to strtof(), except that the conversion result is of type `long double` instead of `float`.

strtold() is currently only supported when `long double` is a 32-bit type.

21.10.4.37 strtoul()

```
unsigned long strtoul (
    const char * __nptr,
    char ** __endptr,
    int __base )
```

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

The string may begin with an arbitrary amount of white space (as determined by isspace()) followed by a single optional `'+'` or `'-'` sign. If `base` is zero or 16, the string may then include a `"0x"` prefix, and the number will be read in base 16; otherwise, a zero base is taken as 10 (decimal) unless the next character is `'0'`, in which case it is taken as 8 (octal).

The remainder of the string is converted to an unsigned long value in the obvious manner, stopping at the first character which is not a valid digit in the given base. (In bases above 10, the letter `'A'` in either upper or lower case represents 10, `'B'` represents 11, and so forth, with `'Z'` representing 35.)

If `endptr` is not NULL, strtoul() stores the address of the first invalid character in `*endptr`. If there were no digits at all, however, strtoul() stores the original value of `nptr` in `endptr`. (Thus, if `*nptr` is not `'\0'` but `**endptr` is `'\0'` on return, the entire string was valid.)

The strtoul() function return either the result of the conversion or, if there was a leading minus sign, the negation of the result of the conversion, unless the original (non-negated) value would overflow; in the latter case, strtoul() returns ULONG_MAX, and `errno` is set to ERANGE. If no conversion could be performed, 0 is returned.

21.10.4.38 ultoa()

```
char * ultoa (
    unsigned long val,
    char * s,
    int radix )
```

Convert an unsigned long integer to a string.

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

Note

The minimal size of the buffer `s` depends on the choice of radix. For example, if the radix is 2 (binary), you need to supply a buffer with a minimal length of 8 ∗ sizeof (unsigned long int) + 1 characters, i.e. one character for each bit plus one for the string terminator. Using a larger radix will require a smaller minimal buffer size.

Warning

If the buffer is too small, you risk a buffer overflow.

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

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

21.10.4.39 utoa()

```
char * utoa (
    unsigned int val,
    char * s,
    int radix )
```

Convert an unsigned integer to a string.

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

Note

The minimal size of the buffer `s` depends on the choice of radix. For example, if the radix is 2 (binary), you need to supply a buffer with a minimal length of 8 ∗ sizeof (unsigned int) + 1 characters, i.e. one character for each bit plus one for the string terminator. Using a larger radix will require a smaller minimal buffer size.

Warning

If the buffer is too small, you risk a buffer overflow.

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

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

21.10.5 Variable Documentation

21.10.5.1 __malloc_heap_end `char* __malloc_heap_end [extern]`

`malloc()` tunable.

21.10.5.2 __malloc_heap_start `char* __malloc_heap_start [extern]`

`malloc()` tunable.

21.10.5.3 __malloc_margin `size_t __malloc_margin [extern]`

`malloc()` tunable.

21.11 <string.h>: Strings

Macros

- #define _FFS(x)

Functions

- int ffs (int __val)
- int ffsl (long __val)
- int ffsll (long long __val)
- void ∗ memccpy (void ∗, const void ∗, int, size_t)
- void ∗ memchr (const void ∗, int, size_t)
- int memcmp (const void ∗, const void ∗, size_t)
- void ∗ memcpy (void ∗, const void ∗, size_t)
- void ∗ memmem (const void ∗, size_t, const void ∗, size_t)
- void ∗ memmove (void ∗, const void ∗, size_t)
- void ∗ memrchr (const void ∗, int, size_t)
- void ∗ memset (void ∗, int, size_t)
- char ∗ strcat (char ∗, const char ∗)
- char ∗ strchr (const char ∗, int)
- char ∗ strchrnul (const char ∗, int)
- int strcmp (const char ∗, const char ∗)
- char ∗ strcpy (char ∗, const char ∗)
- int strcasecmp (const char ∗, const char ∗)
- char ∗ strcasestr (const char ∗, const char ∗)
- size_t strcspn (const char ∗__s, const char ∗__reject)
- char ∗ strdup (const char ∗s1)
- char ∗ strndup (const char ∗s, size_t n)
- size_t strlcat (char ∗, const char ∗, size_t)
- size_t strlcpy (char ∗, const char ∗, size_t)
- size_t strlen (const char ∗)
- char ∗ strlwr (char ∗)
- char ∗ strncat (char ∗, const char ∗, size_t)
- int strncmp (const char ∗, const char ∗, size_t)
- char ∗ strncpy (char ∗, const char ∗, size_t)
- int strncasecmp (const char ∗, const char ∗, size_t)
- size_t strnlen (const char ∗, size_t)
- char ∗ strpbrk (const char ∗__s, const char ∗__accept)
- char ∗ strrchr (const char ∗, int)
- char ∗ strrev (char ∗)
- char ∗ strsep (char ∗∗, const char ∗)
- size_t strspn (const char ∗__s, const char ∗__accept)
- char ∗ strstr (const char ∗, const char ∗)
- char ∗ strtok (char ∗, const char ∗)
- char ∗ strtok_r (char ∗, const char ∗, char ∗∗)
- char ∗ strupr (char ∗)

21.11.1 Detailed Description

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

The string functions perform string operations on NULL terminated strings.

Note

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

21.11.2 Macro Definition Documentation

21.11.2.1 _FFS

```
#define _FFS(
    x )
```

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

This macro is very similar to the function ffs() except that it evaluates its argument at compile-time, so it should only be applied to compile-time constant expressions where it will reduce to a constant itself. Application of this macro to expressions that are not constant at compile-time is not recommended, and might result in a huge amount of code generated.

Returns

The _FFS() macro returns the position of the first (least significant) bit set in the word val, or 0 if no bits are set. The least significant bit is position 1. Only 16 bits of argument are evaluted.

21.11.3 Function Documentation

21.11.3.1 ffs()

```
int ffs (
    int val )
```

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

Returns

The ffs() function returns the position of the first (least significant) bit set in the word `val`, or 0 if no bits are set. The least significant bit is position 1.

Note

For expressions that are constant at compile time, consider using the _FFS macro instead.

21.11.3.2 ffsl()

`int ffsl ( long __val )`

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

21.11.3.3 ffsll()

`int ffsll ( long long __val )`

Same as ffs(), for an argument of type `long long`.

21.11.3.4 memccpy()

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

Copy memory area.

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

Returns

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

21.11.3.5 memchr()

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

Scan memory for a character.

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

Returns

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

21.11.3.6 memcmp()

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

Compare memory areas.

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

Returns

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

Note

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

Warning

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

21.11.3.7 memcpy()

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

Copy a memory area.

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

Returns

The memcpy() function returns a pointer to dest.

21.11.3.8 memmem()

```
void * memmem (
    const void * s1,
    size_t len1,
    const void * s2,
    size_t len2 )
```

The memmem() function finds the start of the first occurrence of the substring `s2` of length `len2` in the memory area `s1` of length `len1`.

Returns

The memmem() function returns a pointer to the beginning of the substring, or `NULL` if the substring is not found. If `len2` is zero, the function returns `s1`.

21.11.3.9 memmove()

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

Copy memory area.

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

Returns

The memmove() function returns a pointer to dest.

21.11.3.10 memrchr()

```
void * memrchr (
    const void * src,
    int val,
    size_t len )
```

The memrchr() function is like the memchr() function, except that it searches backwards from the end of the `len` bytes pointed to by `src` instead of forwards from the front. (Glibc, GNU extension.)

Returns

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

21.11.3.11 memset()

```
void * memset (
    void * dest,
    int val,
    size_t len )
```

Fill memory with a constant byte.

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

Returns

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

21.11.3.12 strcasecmp()

```
int strcasecmp (
    const char * s1,
    const char * s2 )
```

Compare two strings ignoring case.

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

Returns

The strcasecmp() function returns an integer less than, equal to, or greater than zero if `s1` is found, respectively, to be less than, to match, or be greater than `s2`. A consequence of the ordering used by strcasecmp() is that if `s1` is an initial substring of `s2`, then `s1` is considered to be "less than" `s2`.

21.11.3.13 strcasestr()

```
char * strcasestr (
    const char * s1,
    const char * s2 )
```

The strcasestr() function finds the first occurrence of the substring `s2` in the string `s1`. This is like strstr(), except that it ignores case of alphabetic symbols in searching for the substring. (Glibc, GNU extension.)

Returns

The strcasestr() function returns a pointer to the beginning of the substring, or `NULL` if the substring is not found. If `s2` points to a string of zero length, the function returns `s1`.

21.11.3.14 strcat()

```
char * strcat (
    char * dest,
    const char * src )
```

Concatenate two strings.

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

Returns

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

21.11.3.15 strchr()

```
char * strchr (
    const char * src,
    int val )
```

Locate character in string.

Returns

The strchr() function returns a pointer to the first occurrence of the character `val` in the string `src`, or `NULL` if the character is not found.

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

21.11.3.16 strchrnul()

```
char * strchrnul (
    const char * s,
    int c )
```

The strchrnul() function is like strchr() except that if `c` is not found in `s`, then it returns a pointer to the null byte at the end of `s`, rather than `NULL`. (Glibc, GNU extension.)

Returns

The strchrnul() function returns a pointer to the matched character, or a pointer to the null byte at the end of `s` (i.e., `s+strlen`(s)) if the character is not found.

21.11.3.17 strcmp()

```
int strcmp (
    const char * s1,
    const char * s2 )
```

Compare two strings.

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

Returns

The strcmp() function returns an integer less than, equal to, or greater than zero if `s1` is found, respectively, to be less than, to match, or be greater than `s2`. A consequence of the ordering used by strcmp() is that if `s1` is an initial substring of `s2`, then `s1` is considered to be "less than" `s2`.

21.11.3.18 strcpy()

```
char * strcpy (
    char * dest,
    const char * src )
```

Copy a string.

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

Returns

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

Note

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

21.11.3.19 strcspn()

```
size_t strcspn (
    const char * s,
    const char * reject )
```

The strcspn() function calculates the length of the initial segment of `s` which consists entirely of characters not in `reject`.

Returns

The strcspn() function returns the number of characters in the initial segment of `s` which are not in the string `reject`. The terminating zero is not considered as a part of string.

21.11.3.20 strdup()

```
char * strdup (
    const char * s1 )
```

Duplicate a string.

The strdup() function allocates memory and copies into it the string addressed by `s1`, including the terminating null character.

Warning

The strdup() function calls malloc() to allocate the memory for the duplicated string! The user is responsible for freeing the memory by calling free().

Returns

The strdup() function returns a pointer to the resulting string dest. If malloc() cannot allocate enough storage for the string, strdup() will return `NULL`.

Warning

Be sure to check the return value of the strdup() function to make sure that the function has succeeded in allocating the memory!

21.11.3.21 strlcat()

```
size_t strlcat (
    char * dst,
    const char * src,
    size_t siz )
```

Concatenate two strings.

Appends `src` to string `dst` of size `siz` (unlike strncat(), `siz` is the full size of `dst`, not space left). At most `siz-1` characters will be copied. Always `'\0'` terminated (unless `siz` <= `strlen(dst)`).

Returns

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

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

Returns

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

21.11.3.22 strlcpy()

```
size_t strlcpy (
    char * dst,
    const char * src,
    size_t siz )
```

Copy a string.

Copy `src` to string `dst` of size `siz`. At most `siz-1` characters will be copied. Always '\0' terminated (unless `siz` == 0).

Returns

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

Copy src to string dst of size siz. At most siz-1 characters will be copied. Always NULL terminates (unless siz == 0).

Returns

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

21.11.3.23 strlen()

```
size_t strlen (
    const char * src )
```

Calculate the length of a string.

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

Returns

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

21.11.3.24 strlwr()

```
char * strlwr (
    char * s )
```

Convert a string to lower case.

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

Returns

The strlwr() function returns a pointer to the converted string. Conversion is perfomed in-place.

21.11.3.25 strncasecmp()

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

Compare two strings ignoring case.

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

Returns

The strncasecmp() function returns an integer less than, equal to, or greater than zero if `s1` (or the first `len` bytes thereof) is found, respectively, to be less than, to match, or be greater than `s2`. A consequence of the ordering used by strncasecmp() is that if `s1` is an initial substring of `s2`, then `s1` is considered to be "less than" `s2`.

21.11.3.26 strncat()

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

Concatenate two strings.

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

Returns

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

21.11.3.27 strncmp()

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

Compare two strings.

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

Returns

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

21.11.3.28 strncpy()

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

Copy a string.

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

In the case where the length of `src` is less than that of `len`, the remainder of `dest` will be padded with nulls (`'\0'`).

Returns

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

21.11.3.29 strndup()

```
char * strndup (
    const char * s,
    size_t len )
```

Duplicate a string.

The strndup() function is similar to strdup(), but copies at most `len` bytes. If `s` is longer than `len`, only `len` bytes are copied, and a terminating null byte (`'\0'`) is added.

Memory for the new string is obtained with malloc(), and can be freed with free().

Returns

The strndup() function returns the location of the newly malloc'ed memory, or `NULL` if the allocation failed.

21.11.3.30 strnlen()

```
size_t strnlen (
    const char * src,
    size_t len )
```

Determine the length of a fixed-size string.

The strnlen() function returns the number of characters in the string pointed to by `src`, not including the terminating '\0' character, but at most `len`. In doing this, strnlen() looks only at the first `len` characters at `src` and never beyond `src` + `len`.

Returns

The strnlen function returns strlen(src), if that is less than `len`, or `len` if there is no '\0' character among the first `len` characters pointed to by `src`.

21.11.3.31 strpbrk()

```
char * strpbrk (
    const char * s,
    const char * accept )
```

The strpbrk() function locates the first occurrence in the string `s` of any of the characters in the string `accept`.

Returns

The strpbrk() function returns a pointer to the character in `s` that matches one of the characters in `accept`, or `NULL` if no such character is found. The terminating zero is not considered as a part of string: if one or both args are empty, the result will be `NULL`.

21.11.3.32 strrchr()

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

Locate character in string.

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

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

Returns

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

21.11.3.33 strrev()

```
char * strrev (
    char * s )
```

Reverse a string.

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

Returns

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

21.11.3.34 strsep()

```
char * strsep (
    char ** sp,
    const char * delim )
```

Parse a string into tokens.

The strsep() function locates, in the string referenced by `*sp`, the first occurrence of any character in the string `delim` (or the terminating '\0' character) and replaces it with a '\0'. The location of the next character after the delimiter character (or `NULL`, if the end of the string was reached) is stored in `*sp`. An ``empty'' field, i.e. one caused by two adjacent delimiter characters, can be detected by comparing the location referenced by the pointer returned in `*sp` to '\0'.

Returns

The strsep() function returns a pointer to the original value of `*sp`. If `*sp` is initially `NULL`, strsep() returns `NULL`.

21.11.3.35 strspn()

```
size_t strspn (
    const char * s,
    const char * accept )
```

The strspn() function calculates the length of the initial segment of `s` which consists entirely of characters in `accept`.

Returns

The strspn() function returns the number of characters in the initial segment of `s` which consist only of characters from `accept`. The terminating zero is not considered as a part of string.

21.11.3.36 strstr()

```
char * strstr (
    const char * s1,
    const char * s2 )
```

Locate a substring.

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

Returns

The strstr() function returns a pointer to the beginning of the substring, or `NULL` if the substring is not found. If `s2` points to a string of zero length, the function returns `s1`.

21.11.3.37 strtok()

```
char * strtok (
    char * s,
    const char * delim )
```

Parses the string s into tokens.

strtok parses the string s into tokens. The first call to strtok should have s as its first argument. Subsequent calls should have the first argument set to `NULL`. If a token ends with a delimiter, this delimiting character is overwritten with a '\0' and a pointer to the next character is saved for the next call to strtok. The delimiter string delim may be different for each call.

Returns

The strtok() function returns a pointer to the next token or `NULL` when no more tokens are found.

Note

strtok() is NOT reentrant. For a reentrant version of this function see `strtok_r()`.

21.11.3.38 strtok_r()

```
char * strtok_r (
    char * string,
    const char * delim,
    char ** last )
```

Parses string into tokens.

strtok_r parses string into tokens. The first call to strtok_r should have string as its first argument. Subsequent calls should have the first argument set to `NULL`. If a token ends with a delimiter, this delimiting character is overwritten with a '\0' and a pointer to the next character is saved for the next call to strtok_r. The delimiter string `delim` may be different for each call. `last` is a user allocated char* pointer. It must be the same while parsing the same string. strtok_r is a reentrant version of strtok().

Returns

The strtok_r() function returns a pointer to the next token or `NULL` when no more tokens are found.

21.11.3.39 strupr()

```
char * strupr (
    char * s )
```

Convert a string to upper case.

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

Returns

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

21.12 <time.h>: Time

Data Structures

- struct tm
- struct week_date

Macros

- #define ONE_HOUR 3600
- #define ONE_DEGREE 3600
- #define ONE_DAY 86400
- #define UNIX_OFFSET 946684800
- #define NTP_OFFSET 3155673600

Typedefs

- typedef uint32_t time_t

Enumerations

- enum _WEEK_DAYS_ {
SUNDAY , **MONDAY** , **TUESDAY** , **WEDNESDAY** ,
THURSDAY , **FRIDAY** , **SATURDAY** }
- enum _MONTHS_ {
JANUARY , **FEBRUARY** , **MARCH** , **APRIL** ,
MAY , **JUNE** , **JULY** , **AUGUST** ,
SEPTEMBER , **OCTOBER** , **NOVEMBER** , **DECEMBER** }

Functions

- time_t time (time_t *timer)
- int32_t difftime (time_t time1, time_t time0)
- time_t mktime (struct tm *timeptr)
- time_t mk_gmtime (const struct tm *timeptr)
- struct tm * gmtime (const time_t *timer)
- void gmtime_r (const time_t *timer, struct tm *timeptr)
- struct tm * localtime (const time_t *timer)
- void localtime_r (const time_t *timer, struct tm *timeptr)
- char * asctime (const struct tm *timeptr)
- void asctime_r (const struct tm *timeptr, char *buf)
- char * ctime (const time_t *timer)
- void ctime_r (const time_t *timer, char *buf)
- char * isotime (const struct tm *tmptr)
- void isotime_r (const struct tm *, char *)
- size_t strftime (char *s, size_t maxsize, const char *format, const struct tm *timeptr)
- void set_dst (int(*)(const time_t *, int32_t *))
- void set_zone (int32_t)
- void set_system_time (time_t timestamp)
- void system_tick (void)
- uint8_t is_leap_year (int16_t year)
- uint8_t month_length (int16_t year, uint8_t month)
- uint8_t week_of_year (const struct tm *timeptr, uint8_t start)
- uint8_t week_of_month (const struct tm *timeptr, uint8_t start)
- struct week_date * iso_week_date (int year, int yday)
- void iso_week_date_r (int year, int yday, struct week_date *)
- uint32_t fatfs_time (const struct tm *timeptr)
- void set_position (int32_t latitude, int32_t longitude)
- int16_t equation_of_time (const time_t *timer)
- int32_t daylight_seconds (const time_t *timer)
- time_t solar_noon (const time_t *timer)
- time_t sun_rise (const time_t *timer)
- time_t sun_set (const time_t *timer)
- float solar_declinationf (const time_t *timer)
- double solar_declination (const time_t *timer)
- long double solar_declinationl (const time_t *timer)
- int8_t moon_phase (const time_t *timer)
- unsigned long gm_sidereal (const time_t *timer)
- unsigned long lm_sidereal (const time_t *timer)

21.12.1 Detailed Description

```
#include <time.h>
```

Introduction to the Time functions This file declares the time functions implemented in AVR-LibC.

The implementation aspires to conform with ISO/IEC 9899 (C90). However, due to limitations of the target processor and the nature of its development environment, a practical implementation must of necessity deviate from the standard.

Section 7.23.2.1 clock() The type clock_t, the macro CLOCKS_PER_SEC, and the function clock() are not implemented. We consider these items belong to operating system code, or to application code when no operating system is present.

Section 7.23.2.3 mktime() The standard specifies that mktime() should return (time_t) -1, if the time cannot be represented. This implementation always returns a 'best effort' representation.

Section 7.23.2.4 time() The standard specifies that time() should return (time_t) -1, if the time is not available. Since the application must initialize the time system, this functionality is not implemented.

Section 7.23.2.2, difftime() Due to the lack of a 64 bit double, the function difftime() returns a long integer. In most cases this change will be invisible to the user, handled automatically by the compiler.

Section 7.23.1.4 struct tm Per the standard, struct tm->tm_isdst is greater than zero when Daylight Saving time is in effect. This implementation further specifies that, when positive, the value of tm_isdst represents the amount time is advanced during Daylight Saving time.

Section 7.23.3.5 strftime() Only the 'C' locale is supported, therefore the modifiers 'E' and 'O' are ignored. The 'Z' conversion is also ignored, due to the lack of time zone name.

In addition to the above departures from the standard, there are some behaviors which are different from what is often expected, though allowed under the standard.

There is no 'platform standard' method to obtain the current time, time zone, or daylight savings 'rules' in the AVR environment. Therefore the application must initialize the time system with this information. The functions set_zone(), set_dst(), and set_system_time() are provided for initialization. Once initialized, system time is maintained by calling the function system_tick() at one second intervals.

Though not specified in the standard, it is often expected that time_t is a signed integer representing an offset in seconds from Midnight Jan 1 1970... i.e. 'Unix time'. This implementation uses an unsigned 32 bit integer offset from Midnight Jan 1 2000. The use of this 'epoch' helps to simplify the conversion functions, while the 32 bit value allows time to be properly represented until Tue Feb 7 06:28:15 2136 UTC. The macros UNIX_OFFSET and NTP_OFFSET are defined to assist in converting to and from Unix and NTP time stamps.

Unlike desktop counterparts, it is impractical to implement or maintain the 'zoneinfo' database. Therefore no attempt is made to account for time zone, daylight saving, or leap seconds in past dates. All calculations are made according to the currently configured time zone and daylight saving 'rule'.

In addition to C standard functions, re-entrant versions of ctime(), asctime(), gmtime() and localtime() are provided which, in addition to being re-entrant, have the property of claiming less permanent storage in RAM. An additional time conversion, isotime() and its re-entrant version, uses far less storage than either ctime() or asctime().

Along with the usual smattering of utility functions, such as is_leap_year(), this library includes a set of functions related the sun and moon, as well as sidereal time functions.

21.12.2 Macro Definition Documentation

21.12.2.1 NTP_OFFSET `#define NTP_OFFSET 3155673600`

Difference between the Y2K and the NTP epochs, in seconds. To convert a Y2K timestamp to NTP...

```
unsigned long ntp;
time_t y2k;

y2k = time(NULL);
ntp = y2k + NTP_OFFSET;
```

21.12.2.2 ONE_DAY `#define ONE_DAY 86400`

One day, expressed in seconds

21.12.2.3 ONE_DEGREE `#define ONE_DEGREE 3600`

Angular degree, expressed in arc seconds

21.12.2.4 ONE_HOUR `#define ONE_HOUR 3600`

One hour, expressed in seconds

21.12.2.5 UNIX_OFFSET `#define UNIX_OFFSET 946684800`

Difference between the Y2K and the UNIX epochs, in seconds. To convert a Y2K timestamp to UNIX...

```
long unix;
time_t y2k;

y2k = time(NULL);
unix = y2k + UNIX_OFFSET;
```

21.12.3 Typedef Documentation

21.12.3.1 time_t `typedef uint32_t time_t`

time_t represents seconds elapsed from Midnight, Jan 1 2000 UTC (the Y2K 'epoch'). Its range allows this implementation to represent time up to Tue Feb 7 06:28:15 2136 UTC.

21.12.4 Enumeration Type Documentation

21.12.4.1 _MONTHS_ `enum _MONTHS_`

Enumerated labels for the months.

21.12.4.2 _WEEK_DAYS_ `enum _WEEK_DAYS_`

Enumerated labels for the days of the week.

21.12.5 Function Documentation

21.12.5.1 asctime()

```
char * asctime (
    const struct tm * timeptr )
```

The asctime function converts the broken-down time of timeptr, into an ascii string in the form

Sun Mar 23 01:03:52 2013

21.12.5.2 asctime_r()

```
void asctime_r (
    const struct tm * timeptr,
    char * buf )
```

Re entrant version of asctime().

21.12.5.3 ctime()

```
char * ctime (
    const time_t * timer )
```

The ctime function is equivalent to asctime(localtime(timer))

21.12.5.4 ctime_r()

```
void ctime_r (
    const time_t * timer,
    char * buf )
```

Re entrant version of ctime().

21.12.5.5 daylight_seconds()

```
int32_t daylight_seconds (
    const time_t * timer )
```

Computes the amount of time the sun is above the horizon, at the location of the observer.

NOTE: At observer locations inside a polar circle, this value can be zero during the winter, and can exceed ONE↩
_DAY during the summer.

The returned value is in seconds.

21.12.5.6 difftime()

```
int32_t difftime (
    time_t time1,
    time_t time0 )
```

The difftime function returns the difference between two binary time stamps, time1 - time0.

21.12.5.7 equation_of_time()

```
int16_t equation_of_time (
        const time_t * timer )
```

Computes the difference between apparent solar time and mean solar time. The returned value is in seconds.

21.12.5.8 fatfs_time()

```
uint32_t fatfs_time (
        const struct tm * timeptr )
```

Convert a Y2K time stamp into a FAT file system time stamp.

21.12.5.9 gm_sidereal()

```
unsigned long gm_sidereal (
        const time_t * timer )
```

Returns Greenwich Mean Sidereal Time, as seconds into the sidereal day. The returned value will range from 0 through 86399 seconds.

21.12.5.10 gmtime()

```
struct tm * gmtime (
        const time_t * timer )
```

The gmtime function converts the time stamp pointed to by timer into broken-down time, expressed as UTC.

21.12.5.11 gmtime_r()

```
void gmtime_r (
        const time_t * timer,
        struct tm * timeptr )
```

Re entrant version of gmtime().

21.12.5.12 is_leap_year()

```
uint8_t is_leap_year (
        int16_t year )
```

Return 1 if year is a leap year, zero if it is not.

21.12.5.13 iso_week_date()

```
struct week_date * iso_week_date (
        int year,
        int yday )
```

Return a week_date structure with the ISO_8601 week based date corresponding to the given year and day of year. See http://en.wikipedia.org/wiki/ISO_week_date for more information.

21.12.5.14 iso_week_date_r()

```
void iso_week_date_r (
        int year,
        int yday,
        struct week_date * iso )
```

Re-entrant version of iso-week_date.

21.12.5.15 isotime()

```
char * isotime (
    const struct tm * tmptr )
```

The isotime function constructs an ascii string in the form
`2013-03-23 01:03:52`

21.12.5.16 isotime_r()

```
void isotime_r (
    const struct tm * tmptr,
    char * buffer )
```

Re entrant version of isotime()

21.12.5.17 lm_sidereal()

```
unsigned long lm_sidereal (
    const time_t * timer )
```

Returns Local Mean Sidereal Time, as seconds into the sidereal day. The returned value will range from 0 through 86399 seconds.

21.12.5.18 localtime()

```
struct tm * localtime (
    const time_t * timer )
```

The localtime function converts the time stamp pointed to by timer into broken-down time, expressed as Local time.

21.12.5.19 localtime_r()

```
void localtime_r (
    const time_t * timer,
    struct tm * timeptr )
```

Re entrant version of localtime().

21.12.5.20 mk_gmtime()

```
time_t mk_gmtime (
    const struct tm * timeptr )
```

This function 'compiles' the elements of a broken-down time structure, returning a binary time stamp. The elements of timeptr are interpreted as representing UTC.

The original values of the tm_wday and tm_yday elements of the structure are ignored, and the original values of the other elements are not restricted to the ranges stated for struct tm.

Unlike mktime(), this function DOES NOT modify the elements of timeptr.

21.12.5.21 mktime()

```
time_t mktime (
    struct tm * timeptr )
```

This function 'compiles' the elements of a broken-down time structure, returning a binary time stamp. The elements of timeptr are interpreted as representing Local Time.

The original values of the tm_wday and tm_yday elements of the structure are ignored, and the original values of the other elements are not restricted to the ranges stated for struct tm.

The element tm_isdst is used for input and output. If set to 0 or a positive value on input, this requests calculation for Daylight Savings Time being off or on, respectively. If set to a negative value on input, it requests calculation to return whether Daylight Savings Time is in effect or not according to the other values.

On successful completion, the values of all elements of timeptr are set to the appropriate range.

21.12.5.22 month_length()

```
uint8_t month_length (
    int16_t year,
    uint8_t month )
```

Return the length of month, given the year and month, where month is in the range 1 to 12.

21.12.5.23 moon_phase()

```
int8_t moon_phase (
    const time_t * timer )
```

Returns an approximation to the phase of the moon. The sign of the returned value indicates a waning or waxing phase. The magnitude of the returned value indicates the percentage illumination.

21.12.5.24 set_dst()

```
void set_dst (
    int(*)(const time_t *, int32_t *) d )
```

Specify the Daylight Saving function.

The Daylight Saving function should examine its parameters to determine whether Daylight Saving is in effect, and return a value appropriate for tm_isdst.

Working examples for the USA and the EU are available..

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

for the European Union, and

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

for the United States

If a Daylight Saving function is not specified, the system will ignore Daylight Saving.

21.12.5.25 set_position()

```
void set_position (
    int32_t latitude,
    int32_t longitude )
```

Set the geographic coordinates of the 'observer', for use with several of the following functions. Parameters are passed as seconds of North Latitude, and seconds of East Longitude.

For New York City...

```
set_position( 40.7142 * ONE_DEGREE, -74.0064 * ONE_DEGREE);
```

21.12.5.26 set_system_time()

```
void set_system_time (
    time_t timestamp )
```

Initialize the system time. Examples are...

From a Clock / Calendar type RTC:

```
struct tm rtc_time;

read_rtc(&rtc_time);
rtc_time.tm_isdst = 0;
set_system_time( mktime(&rtc_time) );
```

From a Network Time Protocol time stamp:

```
set_system_time(ntp_timestamp - NTP_OFFSET);
```

From a UNIX time stamp:

```
set_system_time(unix_timestamp - UNIX_OFFSET);
```

21.12.5.27 set_zone()

```
void set_zone (
    int32_t )
```

Set the 'time zone'. The parameter is given in seconds East of the Prime Meridian. Example for New York City:

```
set_zone(-5 * ONE_HOUR);
```

If the time zone is not set, the time system will operate in UTC only.

21.12.5.28 solar_declination()

```
double solar_declination (
    const time_t * timer )
```

Returns the declination of the sun in radians.

This implementation is only available when `double` is a 32-bit type.

21.12.5.29 solar_declinationf()

```
float solar_declinationf (
    const time_t * timer )
```

Returns the declination of the sun in radians.

21.12.5.30 solar_declinationl()

```
long double solar_declinationl (
    const time_t * timer )
```

Returns the declination of the sun in radians.

This implementation is only available when `long double` is a 32-bit type.

21.12.5.31 solar_noon()

```
time_t solar_noon (
    const time_t * timer )
```

Computes the time of solar noon, at the location of the observer.

21.12.5.32 strftime()

```
size_t strftime (
    char * s,
    size_t maxsize,
    const char * format,
    const struct tm * timeptr )
```

A complete description of strftime() is beyond the pale of this document. Refer to ISO/IEC document 9899 for details.

All conversions are made using the 'C Locale', ignoring the E or O modifiers. Due to the lack of a time zone 'name', the 'Z' conversion is also ignored.

21.12.5.33 sun_rise()

```
time_t sun_rise (
    const time_t * timer )
```

Return the time of sunrise, at the location of the observer. See the note about daylight_seconds().

21.12.5.34 sun_set()

```
time_t sun_set (
    const time_t * timer )
```

Return the time of sunset, at the location of the observer. See the note about daylight_seconds().

21.12.5.35 system_tick()

```
void system_tick (
    void )
```

Maintain the system time by calling this function at a rate of 1 Hertz.

It is anticipated that this function will typically be called from within an Interrupt Service Routine, (though that is not required). It therefore includes code which makes it simple to use from within a 'Naked' ISR, avoiding the cost of saving and restoring all the cpu registers.

Such an ISR may resemble the following example...

```
ISR(RTC_OVF_vect, ISR_NAKED)
{
    system_tick();
    reti();
}
```

21.12.5.36 time()

```
time_t time (
    time_t * timer )
```

The time function returns the systems current time stamp. If timer is not a null pointer, the return value is also assigned to the object it points to.

21.12.5.37 week_of_month()

```
uint8_t week_of_month (
    const struct tm * timeptr,
    uint8_t start )
```

Return the calendar week of month, where the first week is considered to begin on the day of week specified by 'start'. The returned value may range from zero to 5.

21.12.5.38 week_of_year()

```
uint8_t week_of_year (
    const struct tm * timeptr,
    uint8_t start )
```

Return the calendar week of year, where week 1 is considered to begin on the day of week specified by 'start'. The returned value may range from zero to 52.

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

Macros

- #define BOOTLOADER_SECTION __attribute__ ((__section__(".bootloader")))
- #define boot_spm_interrupt_enable() (__SPM_REG |= (uint8_t)_BV(SPMIE))
- #define boot_spm_interrupt_disable() (__SPM_REG &= (uint8_t)~_BV(SPMIE))
- #define boot_is_spm_interrupt() (__SPM_REG & (uint8_t)_BV(SPMIE))
- #define boot_rww_busy() (__SPM_REG & (uint8_t)_BV(__COMMON_ASB))
- #define boot_spm_busy() (__SPM_REG & (uint8_t)_BV(__SPM_ENABLE))
- #define boot_spm_busy_wait() do{}while(boot_spm_busy())
- #define GET_LOW_FUSE_BITS (0x0000)
- #define GET_LOCK_BITS (0x0001)
- #define GET_EXTENDED_FUSE_BITS (0x0002)
- #define GET_HIGH_FUSE_BITS (0x0003)
- #define boot_lock_fuse_bits_get(address)
- #define boot_signature_byte_get(addr)
- #define boot_page_fill(address, data) __boot_page_fill_normal(address, data)
- #define boot_page_erase(address) __boot_page_erase_normal(address)
- #define boot_page_write(address) __boot_page_write_normal(address)
- #define boot_rww_enable() __boot_rww_enable()
- #define boot_lock_bits_set(lock_bits) __boot_lock_bits_set(lock_bits)
- #define boot_page_fill_safe(address, data)
- #define boot_page_erase_safe(address)
- #define boot_page_write_safe(address)
- #define boot_rww_enable_safe()
- #define boot_lock_bits_set_safe(lock_bits)

21.13.1 Detailed Description

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

The macros in this module provide a C language interface to the bootloader support functionality of certain AVR processors. These macros are designed to work with all sizes of flash memory.

Global interrupts are not automatically disabled for these macros. It is left up to the programmer to do this. See the code example below. Also see the processor datasheet for caveats on having global interrupts enabled during writing of the Flash.

Note

> Not all AVR processors provide bootloader support. See your processor datasheet to see if it provides bootloader support.

API Usage Example

The following code shows typical usage of the boot API.

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

void boot_program_page (uint32_t page, uint8_t *buf)
{
    // Disable interrupts.
    uint8_t sreg = SREG;
    cli();

    eeprom_busy_wait ();

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

    for (uint16_t i = 0; i < SPM_PAGESIZE; i += 2)
    {
        // Set up little-endian word.
        uint16_t w = *buf++;
        w += (*buf++) << 8;

        boot_page_fill (page + i, w);
    }

    boot_page_write (page);     // Store buffer in flash page.
    boot_spm_busy_wait();       // Wait until the memory is written.

    // Reenable RWW-section again. We need this if we want to jump back
    // to the application after bootloading.
    boot_rww_enable ();

    // Re-enable interrupts (if they were ever enabled).
    SREG = sreg;
}
```

21.13.2 Macro Definition Documentation

21.13.2.1 boot_is_spm_interrupt

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

Check if the SPM interrupt is enabled.

21.13.2.2 boot_lock_bits_set

```
#define boot_lock_bits_set(
        lock_bits ) __boot_lock_bits_set(lock_bits)
```

Set the bootloader lock bits.

Parameters

lock_bits	A mask of which Boot Loader Lock Bits to set.

Note

In this context, a 'set bit' will be written to a zero value. Note also that only BLBxx bits can be programmed by this command.

For example, to disallow the SPM instruction from writing to the Boot Loader memory section of flash, you would use this macro as such:

```
boot_lock_bits_set (_BV (BLB11));
```

Note

Like any lock bits, the Boot Loader Lock Bits, once set, cannot be cleared again except by a chip erase which will in turn also erase the boot loader itself.

21.13.2.3 boot_lock_bits_set_safe #define boot_lock_bits_set_safe(*lock_bits*)

Value:
```
do { \
    boot_spm_busy_wait();                          \
    eeprom_busy_wait();                            \
    boot_lock_bits_set (lock_bits);                \
} while (0)
```

Same as boot_lock_bits_set() except waits for eeprom and spm operations to complete before setting the lock bits.

21.13.2.4 boot_lock_fuse_bits_get #define boot_lock_fuse_bits_get(*address*)

Value:
```
(({                                                \
    uint8_t __result;                                    \
    __asm__ __volatile__                                 \
    (                                                    \
        "sts %1, %2\n\t"                                 \
        "lpm %0, Z\n\t"                                  \
        : "=r" (__result)                                \
        : "i" (_SFR_MEM_ADDR(__SPM_REG)),                \
          "r" ((uint8_t)(__BOOT_LOCK_BITS_SET)),         \
          "z" ((uint16_t)(address))                      \
    );                                                   \
    __result;                                            \
}))
```

Read the lock or fuse bits at `address`.

Parameter `address` can be any of GET_LOW_FUSE_BITS, GET_LOCK_BITS, GET_EXTENDED_FUSE_BITS, or GET_HIGH_FUSE_BITS.

Note

The lock and fuse bits returned are the physical values, i.e. a bit returned as 0 means the corresponding fuse or lock bit is programmed.

21.13.2.5 boot_page_erase #define boot_page_erase(*address*) __boot_page_erase_normal(address)

Erase the flash page that contains address.

Note

address is a byte address in flash, not a word address.

21.13.2.6 boot_page_erase_safe `#define boot_page_erase_safe( address )`

Value:
```
do { \
    boot_spm_busy_wait();                    \
    eeprom_busy_wait();                      \
    boot_page_erase (address);               \
} while (0)
```

Same as boot_page_erase() except it waits for eeprom and spm operations to complete before erasing the page.

21.13.2.7 boot_page_fill `#define boot_page_fill( address, data ) __boot_page_fill_normal(address, data)`

Fill the bootloader temporary page buffer for flash address with data word.

Note

The address is a byte address. The data is a word. The AVR writes data to the buffer a word at a time, but addresses the buffer per byte! So, increment your address by 2 between calls, and send 2 data bytes in a word format! The LSB of the data is written to the lower address; the MSB of the data is written to the higher address.

21.13.2.8 boot_page_fill_safe `#define boot_page_fill_safe( address, data )`

Value:
```
do { \
    boot_spm_busy_wait();                    \
    eeprom_busy_wait();                      \
    boot_page_fill(address, data);           \
} while (0)
```

Same as boot_page_fill() except it waits for eeprom and spm operations to complete before filling the page.

21.13.2.9 boot_page_write `#define boot_page_write( address ) __boot_page_write_normal(address)`

Write the bootloader temporary page buffer to flash page that contains address.

Note

address is a byte address in flash, not a word address.

21.13.2.10 boot_page_write_safe `#define boot_page_write_safe( address )`

Value:
```
do { \
    boot_spm_busy_wait();                      \
    eeprom_busy_wait();                        \
    boot_page_write (address);                 \
} while (0)
```

Same as boot_page_write() except it waits for eeprom and spm operations to complete before writing the page.

21.13.2.11 boot_rww_busy `#define boot_rww_busy( ) (__SPM_REG & (uint8_t)_BV(__COMMON_ASB))`

Check if the RWW section is busy.

21.13.2.12 boot_rww_enable `#define boot_rww_enable( ) __boot_rww_enable()`

Enable the Read-While-Write memory section.

21.13.2.13 boot_rww_enable_safe `#define boot_rww_enable_safe( )`

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

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

21.13.2.14 boot_signature_byte_get `#define boot_signature_byte_get( addr )`

Value:
```
  (({                              \
      uint8_t __result;                        \
      __asm__ __volatile__                     \
      (                                        \
        "sts %1, %2"    "\n\t"                 \
        "lpm %0, Z"                            \
        : "=r" (__result)                      \
        : "i" (_SFR_MEM_ADDR(__SPM_REG)),      \
          "r" ((uint8_t)(__BOOT_SIGROW_READ)), \
          "z" ((uint16_t)(addr))               \
      );                                       \
      __result;                                \
}))
```

Read the Signature Row byte at `address`. For some MCU types, this function can also retrieve the factory-stored oscillator calibration bytes.

Parameter `address` can be 0-0x1f as documented by the datasheet.

Note

The values are MCU type dependent.

21.13.2.15 boot_spm_busy `#define boot_spm_busy( ) (__SPM_REG & (uint8_t)_BV(__SPM_ENABLE))`

Check if the SPM instruction is busy.

21.13.2.16 boot_spm_busy_wait `#define boot_spm_busy_wait( ) do{}while(boot_spm_busy())`

Wait while the SPM instruction is busy.

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

Disable the SPM interrupt.

21.13.2.18 boot_spm_interrupt_enable `#define boot_spm_interrupt_enable( ) (__SPM_REG |= (uint8_t)_BV(SPMIE))`

Enable the SPM interrupt.

21.13.2.19 BOOTLOADER_SECTION `#define BOOTLOADER_SECTION __attribute__ ((__section__↩ (".bootloader")))`

Used to declare a function or variable to be placed into a new section called .bootloader. This section and its contents can then be relocated to any address (such as the bootloader NRWW area) at link-time.

21.13.2.20 GET_EXTENDED_FUSE_BITS `#define GET_EXTENDED_FUSE_BITS (0x0002)`

address to read the extended fuse bits, using boot_lock_fuse_bits_get

21.13.2.21 GET_HIGH_FUSE_BITS `#define GET_HIGH_FUSE_BITS (0x0003)`

address to read the high fuse bits, using boot_lock_fuse_bits_get

21.13.2.22 GET_LOCK_BITS `#define GET_LOCK_BITS (0x0001)`

address to read the lock bits, using boot_lock_fuse_bits_get

21.13.2.23 GET_LOW_FUSE_BITS `#define GET_LOW_FUSE_BITS (0x0000)`

address to read the low fuse bits, using boot_lock_fuse_bits_get

21.14 <avr/cpufunc.h>: Special AVR CPU functions

Macros

- #define _NOP()
- #define _MemoryBarrier()

Functions

- void ccp_write_io (volatile uint8_t *__ioaddr, uint8_t __value)
- void ccp_write_spm (volatile uint8_t *__ioaddr, uint8_t __value)

21.14.1 Detailed Description

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

This header file contains macros that access special functions of the AVR CPU which do not fit into any of the other header files.

21.14.2 Macro Definition Documentation

21.14.2.1 _MemoryBarrier `#define _MemoryBarrier( )`

Implement a read/write *memory barrier*. A memory barrier instructs the compiler to not cache any memory data in registers beyond the barrier. This can sometimes be more effective than blocking certain optimizations by declaring some object with a `volatile` qualifier.

See Problems with reordering code for things to be taken into account with respect to compiler optimizations.

21.14.2.2 _NOP `#define _NOP( )`

Execute a *no operation* (NOP) CPU instruction. This should not be used to implement delays, better use the functions from <util/delay_basic.h> or <util/delay.h> for this. For debugging purposes, a NOP can be useful to have an instruction that is guaranteed to be not optimized away by the compiler, so it can always become a breakpoint in the debugger.

21.14.3 Function Documentation

21.14.3.1 ccp_write_io()

```
void ccp_write_io (
    volatile uint8_t * __ioaddr,
    uint8_t __value )
```

Write *__value* to IO Register Protected (CCP) IO register at *__ioaddr*. . See also `_PROTECTED_WRITE()`.

21.14.3.2 ccp_write_spm()

```
void ccp_write_spm (
    volatile uint8_t * __ioaddr,
    uint8_t __value )
```

Write *__value* to SPM Instruction Protected (CCP) IO register at *__ioaddr*. See also `_PROTECTED_WRITE_SPM()`.

21.15 <avr/eeprom.h>: EEPROM handling

Macros

- #define EEMEM __attribute__((__section__(".eeprom")))
- #define eeprom_is_ready()
- #define eeprom_busy_wait() do {} while (!eeprom_is_ready())

Functions

- uint8_t eeprom_read_byte (const uint8_t *__p)
- uint16_t eeprom_read_word (const uint16_t *__p)
- uint32_t eeprom_read_dword (const uint32_t *__p)
- uint64_t eeprom_read_qword (const uint64_t *__p)
- float eeprom_read_float (const float *__p)
- double eeprom_read_double (const double *__p)
- long double eeprom_read_long_double (const long double *__p)
- void eeprom_read_block (void *__dst, const void *__src, size_t __n)
- void eeprom_write_byte (uint8_t *__p, uint8_t __value)
- void eeprom_write_word (uint16_t *__p, uint16_t __value)
- void eeprom_write_dword (uint32_t *__p, uint32_t __value)
- void eeprom_write_qword (uint64_t *__p, uint64_t __value)
- void eeprom_write_float (float *__p, float __value)
- void eeprom_write_double (double *__p, double __value)
- void eeprom_write_long_double (long double *__p, long double __value)
- void eeprom_write_block (const void *__src, void *__dst, size_t __n)
- void eeprom_update_byte (uint8_t *__p, uint8_t __value)
- void eeprom_update_word (uint16_t *__p, uint16_t __value)
- void eeprom_update_dword (uint32_t *__p, uint32_t __value)
- void eeprom_update_qword (uint64_t *__p, uint64_t __value)
- void eeprom_update_float (float *__p, float __value)
- void eeprom_update_double (double *__p, double __value)
- void eeprom_update_long_double (long double *__p, long double __value)
- void eeprom_update_block (const void *__src, void *__dst, size_t __n)

IAR C compatibility defines

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

21.15.1 Detailed Description

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

This header file declares the interface to some simple library routines suitable for handling the data EEPROM contained in the AVR microcontrollers. The implementation uses a simple polled mode interface. Applications that require interrupt-controlled EEPROM access to ensure that no time will be wasted in spinloops will have to deploy their own implementation.

Notes:

- In addition to the write functions there is a set of update ones. This functions read each byte first and skip the burning if the old value is the same with new. The scaning direction is from high address to low, to obtain quick return in common cases.
- All of the read/write functions first make sure the EEPROM is ready to be accessed. Since this may cause long delays if a write operation is still pending, time-critical applications should first poll the EEPROM e. g. using eeprom_is_ready() before attempting any actual I/O. But this functions does not wait until SELFPRGEN in SPMCSR becomes zero. Do this manually, if your softwate contains the Flash burning.
- As these functions modify IO registers, they are known to be non-reentrant. If any of these functions are used from both, standard and interrupt context, the applications must ensure proper protection (e.g. by disabling interrupts before accessing them).
- All write functions force erase_and_write programming mode.
- For Xmega the EEPROM start address is 0, like other architectures. The reading functions add the 0x2000 value to use EEPROM mapping into data space.

21.15.2 Macro Definition Documentation

21.15.2.1 __EEGET

```
#define __EEGET(
    var,
    addr ) (var) = eeprom_read_byte ((const uint8_t *)(addr))
```

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

21.15.2.2 __EEPUT

```
#define __EEPUT(
    addr,
    val ) eeprom_write_byte ((uint8_t *)(addr), (uint8_t)(val))
```

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

21.15.2.3 _EEGET

```
#define _EEGET(
    var,
    addr ) (var) = eeprom_read_byte ((const uint8_t *)(addr))
```

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

21.15.2.4 _EEPUT `#define _EEPUT( addr, val ) eeprom_write_byte ((uint8_t *)(addr), (uint8_t)(val))`

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

21.15.2.5 EEMEM `#define EEMEM __attribute__((__section__(".eeprom")))`

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

21.15.2.6 eeprom_busy_wait `#define eeprom_busy_wait( ) do {} while (!eeprom_is_ready())`

Loops until the eeprom is no longer busy.

Returns

Nothing.

21.15.2.7 eeprom_is_ready `#define eeprom_is_ready( )`

Returns

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

21.15.3 Function Documentation

21.15.3.1 eeprom_read_block() `void eeprom_read_block ( void * __dst, const void * __src, size_t __n )`

Read a block of *__n* bytes from EEPROM address *__src* to SRAM *__dst*.

21.15.3.2 eeprom_read_byte() `uint8_t eeprom_read_byte ( const uint8_t * __p )`

Read one byte from EEPROM address *__p*.

21.15.3.3 eeprom_read_double() `double eeprom_read_double ( const double * __p )`

Read one double value (little endian) from EEPROM address *__p*.

21.15.3.4 eeprom_read_dword()

```
uint32_t eeprom_read_dword (
    const uint32_t * __p )
```

Read one 32-bit double word (little endian) from EEPROM address *__p*.

21.15.3.5 eeprom_read_float()

```
float eeprom_read_float (
    const float * __p )
```

Read one float value (little endian) from EEPROM address *__p*.

21.15.3.6 eeprom_read_long_double()

```
long double eeprom_read_long_double (
    const long double * __p )
```

Read one long double value (little endian) from EEPROM address *__p*.

21.15.3.7 eeprom_read_qword()

```
uint64_t eeprom_read_qword (
    const uint64_t * __p )
```

Read one 64-bit quad word (little endian) from EEPROM address *__p*.

21.15.3.8 eeprom_read_word()

```
uint16_t eeprom_read_word (
    const uint16_t * __p )
```

Read one 16-bit word (little endian) from EEPROM address *__p*.

21.15.3.9 eeprom_update_block()

```
void eeprom_update_block (
    const void * __src,
    void * __dst,
    size_t __n )
```

Update a block of *__n* bytes at EEPROM address *__dst* from *__src*.

Note

The argument order is mismatch with common functions like strcpy().

21.15.3.10 eeprom_update_byte()

```
void eeprom_update_byte (
    uint8_t * __p,
    uint8_t __value )
```

Update a byte *__value* at EEPROM address *__p*.

21.15.3.11 eeprom_update_double()

```
void eeprom_update_double (
    double * __p,
    double __value )
```

Update a double *__value* at EEPROM address *__p*.

21.15.3.12 eeprom_update_dword()

```
void eeprom_update_dword (
    uint32_t * __p,
    uint32_t __value )
```

Update a 32-bit double word *__value* at EEPROM address *__p*.

21.15.3.13 eeprom_update_float()

```
void eeprom_update_float (
    float * __p,
    float __value )
```

Update a float *__value* at EEPROM address *__p*.

21.15.3.14 eeprom_update_long_double()

```
void eeprom_update_long_double (
    long double * __p,
    long double __value )
```

Update a long double *__value* at EEPROM address *__p*.

21.15.3.15 eeprom_update_qword()

```
void eeprom_update_qword (
    uint64_t * __p,
    uint64_t __value )
```

Update a 64-bit quad word *__value* at EEPROM address *__p*.

21.15.3.16 eeprom_update_word()

```
void eeprom_update_word (
    uint16_t * __p,
    uint16_t __value )
```

Update a word *__value* at EEPROM address *__p*.

21.15.3.17 eeprom_write_block()

```
void eeprom_write_block (
    const void * __src,
    void * __dst,
    size_t __n )
```

Write a block of *__n* bytes to EEPROM address *__dst* from *__src*.

Note

The argument order is mismatch with common functions like strcpy().

21.15.3.18 eeprom_write_byte()

```
void eeprom_write_byte (
    uint8_t * __p,
    uint8_t __value )
```

Write a byte *__value* to EEPROM address *__p*.

21.15.3.19 eeprom_write_double()

```
void eeprom_write_double (
    double * __p,
    double __value )
```

Write a double *__value* to EEPROM address *__p*.

21.15.3.20 eeprom_write_dword()

```
void eeprom_write_dword (
    uint32_t * __p,
    uint32_t __value )
```

Write a 32-bit double word *__value* to EEPROM address *__p*.

21.15.3.21 eeprom_write_float()

```
void eeprom_write_float (
    float * __p,
    float __value )
```

Write a float *__value* to EEPROM address *__p*.

21.15.3.22 eeprom_write_long_double()

```
void eeprom_write_long_double (
    long double * __p,
    long double __value )
```

Write a long double *__value* to EEPROM address *__p*.

21.15.3.23 eeprom_write_qword()

```
void eeprom_write_qword (
    uint64_t * __p,
    uint64_t __value )
```

Write a 64-bit quad word *__value* to EEPROM address *__p*.

21.15.3.24 eeprom_write_word()

```
void eeprom_write_word (
    uint16_t * __p,
    uint16_t __value )
```

Write a word *__value* to EEPROM address *__p*.

21.16 <avr/fuse.h>: Fuse Support

Introduction

The Fuse API allows a user to specify the fuse settings for the specific AVR device they are compiling for. These fuse settings will be placed in a special section in the ELF output file, after linking.

Programming tools can take advantage of the fuse information embedded in the ELF file, by extracting this information and determining if the fuses need to be programmed before programming the Flash and EEPROM memories. This also allows a single ELF file to contain all the information needed to program an AVR.

To use the Fuse API, include the <avr/io.h> header file, which in turn automatically includes the individual I/O header file and the <avr/fuse.h> file. These other two files provides everything necessary to set the AVR fuses.

Fuse API

Each I/O header file must define the FUSE_MEMORY_SIZE macro which is defined to the number of fuse bytes that exist in the AVR device.

A new type, __fuse_t, is defined as a structure. The number of fields in this structure are determined by the number of fuse bytes in the FUSE_MEMORY_SIZE macro.

If FUSE_MEMORY_SIZE == 1, there is only a single field: byte, of type unsigned char.

If FUSE_MEMORY_SIZE == 2, there are two fields: low, and high, of type unsigned char.

If FUSE_MEMORY_SIZE == 3, there are three fields: low, high, and extended, of type unsigned char.

If FUSE_MEMORY_SIZE > 3, there is a single field: byte, which is an array of unsigned char with the size of the array being FUSE_MEMORY_SIZE.

A convenience macro, FUSEMEM, is defined as a GCC attribute for a custom-named section of ".fuse".

A convenience macro, FUSES, is defined that declares a variable, __fuse, of type __fuse_t with the attribute defined by FUSEMEM. This variable allows the end user to easily set the fuse data.

Note

> If a device-specific I/O header file has previously defined FUSEMEM, then FUSEMEM is not redefined. If a device-specific I/O header file has previously defined FUSES, then FUSES is not redefined.

Each AVR device I/O header file has a set of defined macros which specify the actual fuse bits available on that device. The AVR fuses have inverted values, logical 1 for an unprogrammed (disabled) bit and logical 0 for a programmed (enabled) bit. The defined macros for each individual fuse bit represent this in their definition by a bit-wise inversion of a mask. For example, the FUSE_EESAVE fuse in the ATmega128 is defined as:

```
#define FUSE_EESAVE      ~_BV(3)
```

Note

> The _BV macro creates a bit mask from a bit number. It is then inverted to represent logical values for a fuse memory byte.

To combine the fuse bits macros together to represent a whole fuse byte, use the bitwise AND operator, like so:

```
(FUSE_BOOTSZ0 & FUSE_BOOTSZ1 & FUSE_EESAVE & FUSE_SPIEN & FUSE_JTAGEN)
```

Each device I/O header file also defines macros that provide default values for each fuse byte that is available. LFUSE_DEFAULT is defined for a Low Fuse byte. HFUSE_DEFAULT is defined for a High Fuse byte. EFUSE_↩ DEFAULT is defined for an Extended Fuse byte.

If FUSE_MEMORY_SIZE > 3, then the I/O header file defines macros that provide default values for each fuse byte like so: FUSE0_DEFAULT FUSE1_DEFAULT FUSE2_DEFAULT FUSE3_DEFAULT FUSE4_DEFAULT ....

API Usage Example

Putting all of this together is easy. Using C99's designated initializers:

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

FUSES =
{
    .low = LFUSE_DEFAULT,
    .high = (FUSE_BOOTSZ0 & FUSE_BOOTSZ1 & FUSE_EESAVE & FUSE_SPIEN & FUSE_JTAGEN),
    .extended = EFUSE_DEFAULT,
};

int main(void)
{
    return 0;
}
```

Or, using the variable directly instead of the FUSES macro,

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

__fuse_t __fuse __attribute__((section (".fuse"))) =
{
    .low = LFUSE_DEFAULT,
    .high = (FUSE_BOOTSZ0 & FUSE_BOOTSZ1 & FUSE_EESAVE & FUSE_SPIEN & FUSE_JTAGEN),
    .extended = EFUSE_DEFAULT,
};

int main(void)
{
    return 0;
}
```

If you are compiling in C++, you cannot use the designated intializers so you must do:

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

FUSES =
{
    LFUSE_DEFAULT, // .low
    (FUSE_BOOTSZ0 & FUSE_BOOTSZ1 & FUSE_EESAVE & FUSE_SPIEN & FUSE_JTAGEN), // .high
    EFUSE_DEFAULT, // .extended
};

int main(void)
{
    return 0;
}
```

However there are a number of caveats that you need to be aware of to use this API properly.

Be sure to include <avr/io.h> to get all of the definitions for the API. The FUSES macro defines a global variable to store the fuse data. This variable is assigned to its own linker section. Assign the desired fuse values immediately in the variable initialization.

The .fuse section in the ELF file will get its values from the initial variable assignment ONLY. This means that you can NOT assign values to this variable in functions and the new values will not be put into the ELF .fuse section.

The global variable is declared in the FUSES macro has two leading underscores, which means that it is reserved for the "implementation", meaning the library, so it will not conflict with a user-named variable.

You must initialize ALL fields in the __fuse_t structure. This is because the fuse bits in all bytes default to a logical 1, meaning unprogrammed. Normal uninitialized data defaults to all locgial zeros. So it is vital that all fuse bytes are initialized, even with default data. If they are not, then the fuse bits may not programmed to the desired settings.

Be sure to have the -mmcu=*device* flag in your compile command line and your linker command line to have the correct device selected and to have the correct I/O header file included when you include <avr/io.h>.

You can print out the contents of the .fuse section in the ELF file by using this command line:

```
avr-objdump -s -j .fuse <ELF file>
```

The section contents shows the address on the left, then the data going from lower address to a higher address, left to right.

21.17 <avr/interrupt.h>: Interrupts

Global manipulation of the interrupt flag

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

Handling interrupts frequently requires attention regarding atomic access to objects that could be altered by code running within an interrupt context, see <util/atomic.h>.

Frequently, interrupts are being disabled for periods of time in order to perform certain operations without being disturbed; see Problems with reordering code for things to be taken into account with respect to compiler optimizations.

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

Macros for writing interrupt handler functions

- #define ISR(vector, attributes)
- #define SIGNAL(vector)
- #define EMPTY_INTERRUPT(vector)
- #define ISR_ALIAS(vector, target_vector)
- #define reti() __asm__ __volatile__ ("reti" ::: "memory")
- #define BADISR_vect

ISR attributes

- #define ISR_BLOCK
- #define ISR_NOBLOCK
- #define ISR_NAKED
- #define ISR_FLATTEN
- #define ISR_NOICF
- #define ISR_NOGCCISR
- #define ISR_ALIASOF(target_vector)

21.17.1 Detailed Description

Note

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

Introduction to AVR-LibC's interrupt handling It's nearly impossible to find compilers that agree on how to handle interrupt code. Since the C language tries to stay away from machine dependent details, each compiler writer is forced to design their method of support.

In the AVR-GCC environment, the vector table is predefined to point to interrupt routines with predetermined names. By using the appropriate name, your routine will be called when the corresponding interrupt occurs. The device library provides a set of default interrupt routines, which will get used if you don't define your own.

Patching into the vector table is only one part of the problem. The compiler uses, by convention, a set of registers when it's normally executing compiler-generated code. It's important that these registers, as well as the status register, get saved and restored.

These details seem to make interrupt routines a little messy, but all these details are handled by the Interrupt API. An interrupt routine is defined with ISR(). This macro register and mark the routine as an interrupt handler for the specified peripheral. The following is an example definition of a handler for the ADC interrupt.

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

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

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

Catch-all interrupt vector If an unexpected interrupt occurs (interrupt is enabled but no handler is installed, which usually indicates a bug), then the default action is to reset the device by jumping to the reset vector. You can override this by supplying a function named `BADISR_vect` which should be defined with ISR() as such. The name `BADISR_vect` is actually an alias for `__vector_default`. The latter must be used inside assembly code in case <avr/interrupt.h> is not included.

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

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

Nested interrupts The AVR hardware clears the global interrupt flag in SREG when an interrupt request is serviced. Thus, normally interrupts will remain disabled inside the handler until the handler exits, where the RETI instruction (that is emitted by the compiler as part of the normal function epilogue for an ISR) will eventually re-enable further interrupts. For that reason, interrupt handlers normally do not nest. For most interrupt handlers, this is the desired behaviour, for some it is even required in order to prevent infinitely recursive interrupts (like UART interrupts, or level-triggered external interrupts). In rare circumstances though it might be desired to re-enable the global interrupt flag as early as possible in the interrupt handler, in order to not defer any other interrupt more than absolutely needed. This could be done using an `sei()` instruction right at the beginning of the interrupt handler, but this still leaves few instructions inside the compiler-generated function prologue to run with global interrupts disabled. The compiler can be instructed to insert a `SEI` instruction right at the beginning of an interrupt handler by declaring the handler the following way:

```
ISR(XXX_vect, ISR_NOBLOCK)
{
  ...
}
```

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

Two vectors sharing the same code In some circumstances, the actions to be taken upon two different interrupts might be completely identical so a single implementation for the ISR would suffice. For example, pin-change interrupts arriving from two different ports could logically signal an event that is independent from the actual port (and thus interrupt vector) where it happened. Sharing interrupt vector code can be accomplished using the ISR_ALIASOF() attribute to the ISR macro:

```
ISR(PCINT0_vect)
{
  ...
  // Code to handle the event.
}

ISR(PCINT1_vect, ISR_ALIASOF(PCINT0_vect));
```

Note

There is no body to the aliased ISR.

Note that the ISR_ALIASOF() feature requires GCC 4.2 or above (or a patched version of GCC 4.1.x). See the documentation of the ISR_ALIAS() macro for an implementation which is less elegant but could be applied to all compiler versions.

Empty interrupt service routines In rare circumstances, in interrupt vector does not need any code to be implemented at all. The vector must be declared anyway, so when the interrupt triggers it won't execute the BADISR_vect code (which by default restarts the application).

This could for example be the case for interrupts that are solely enabled for the purpose of getting the controller out of sleep_mode().

A handler for such an interrupt vector can be declared using the EMPTY_INTERRUPT() macro:

```
EMPTY_INTERRUPT(ADC_vect);
```

Note

There is no body to this macro.

Manually defined ISRs In some circumstances, the compiler-generated prologue and epilogue of the ISR might not be optimal for the job, and a manually defined ISR could be considered particularly to speedup the interrupt handling.

One solution to this could be to implement the entire ISR as manual assembly code in a separate (assembly) file. See Combining C and assembly source files for an example of how to implement it that way.

Another solution is to still implement the ISR in C language but take over the compiler's job of generating the prologue and epilogue. This can be done using the ISR_NAKED attribute to the ISR() macro. Note that the compiler does not generate *anything* as prologue or epilogue, so the final reti() must be provided by the actual implementation. SREG must be manually saved if the ISR code modifies it, and the compiler-implied assumption of `__zero_reg__` always being 0 could be wrong (e. g. when interrupting right after of a `MUL` instruction).

Warning

According to the GCC documentation, only inline assembly is supported in `naked` functions, like with ISR_NAKED.

```
ISR(TIMER1_OVF_vect,  ISR_NAKED)
{
  PORTB |= _BV(0);  // results in SBI which does not affect SREG
  reti();
}
```

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

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

- Starting with AVR-LibC v1.4, the style of interrupt vector names is a short phrase for the vector description followed by `_vect`. The short phrase matches the vector name as described in the datasheet of the respective device (and in the hardware manufacturer's XML/ATDF files), with spaces replaced by an underscore and other non-alphanumeric characters dropped. Using the suffix `_vect` is intented to improve portability to other C compilers available for the AVR that use a similar naming convention.
- A **deprecated** form that uses names starting with `SIG_`, followed by a relatively verbose but arbitrarily chosen name describing the interrupt vector. This has been the only available style in AVR-LibC up to version 1.2.x. This historical naming style is not recommended for new projects, and some headers require that the macro `__AVR_LIBC_DEPRECATED_ENABLE__` is defined so that the `SIG_` names ISR names are available.

Note

The ISR() macro cannot really spell-check the argument passed to them. Thus, by misspelling one of the names below used in ISR(), a function will be created that, while possibly being usable as an interrupt function, is not actually wired into the interrupt vector table. The compiler will generate a warning if it detects a suspiciously looking name of an ISR() function (i.e. one that after macro replacement does not start with `"__vector_"`).

Apart from the *NAME*`_vect` macros listed below, for each such ISR name there is also a macro *NAME*`_vect_num` defined which resolves to the IRQ number. This is the index into the vector table, where indices start at index 0 (the reset vector).

See also What ISR names are available for my device? in the FAQ for how find all the IRQ names for a specific device.

Table 24 Due to its sheer size, the following table is only available in the HTML version of the documentation.

Vector Name	Description	Applicable for Device

Note

For the following devices, only the deprecated `SIG_` names are available: AT43USB320, AT43USB355, AT76C711, AT90C8534, AT94K, M3000.

21.17.2 Macro Definition Documentation

21.17.2.1 BADISR_vect `#define BADISR_vect`

`#include <avr/interrupt.h>`

This is a vector which is aliased to __vector_default, the vector executed when an IRQ fires with no accompanying ISR handler. This may be used along with the ISR() macro to create a catch-all for undefined but used ISRs for debugging purposes.

21.17.2.2 cli `#define cli( ) __asm__ __volatile__ ("cli" ::: "memory")`

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. However, the macro also implies a *memory barrier* which can cause additional loss of optimization.

In order to implement atomic access to multi-byte objects, consider using the macros from <util/atomic.h>, rather than implementing them manually with cli() and sei().

21.17.2.3 EMPTY_INTERRUPT `#define EMPTY_INTERRUPT( vector )`

Defines an empty interrupt handler function. This will not generate any prolog or epilog code and will only return from the ISR. Do not define a function body as this will define it for you. Example:

```
EMPTY_INTERRUPT(ADC_vect);
```

21.17.2.4 ISR `#define ISR( vector, attributes )`

Introduces an interrupt handler function (interrupt service routine) that runs with global interrupts initially disabled by default with no attributes specified.

The attributes are optional and alter the behaviour and resultant generated code of the interrupt routine. Multiple attributes may be used for a single function, with a space seperating each attribute.

Valid attributes are ISR_BLOCK, ISR_NOBLOCK, ISR_NAKED, ISR_FLATTEN, ISR_NOICF, ISR_NOGCCISR and ISR_ALIASOF(vect).

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

21.17.2.5 ISR_ALIAS `#define ISR_ALIAS( vector, target_vector )`

Aliases a given vector to another one in the same manner as the ISR_ALIASOF attribute for the ISR() macro. Unlike the ISR_ALIASOF attribute macro however, this is compatible for all versions of GCC rather than just GCC version 4.2 onwards.

Note

This macro creates a trampoline function for the aliased macro. This will result in a two cycle penalty for the aliased vector compared to the ISR the vector is aliased to, due to the JMP/RJMP opcode used.

Deprecated For new code, the use of ISR(..., ISR_ALIASOF(...)) is recommended.

Example:

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

ISR_ALIAS(INT1_vect, INT0_vect);
```

21.17.2.6 ISR_ALIASOF `#define ISR_ALIASOF( target_vector )`

The ISR is linked to another ISR, specified by the vect parameter. This is compatible with GCC 4.2 and greater only.

Use this attribute in the attributes parameter of the ISR macro. Example:

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

ISR (INT1_vect, ISR_ALIASOF (INT0_vect));
```

21.17.2.7 ISR_BLOCK `#define ISR_BLOCK`

Identical to an ISR with no attributes specified. Global interrupts are initially disabled by the AVR hardware when entering the ISR, without the compiler modifying this state.

Use this attribute in the attributes parameter of the ISR macro.

21.17.2.8 ISR_FLATTEN `#define ISR_FLATTEN`

The compiler will try to inline all called function into the ISR. This has an effect with GCC 4.6 and newer only.

Use this attribute in the attributes parameter of the ISR macro.

21.17.2.9 ISR_NAKED `#define ISR_NAKED`

ISR is created with no prologue or epilogue code. The user code is responsible for preservation of the machine state including the SREG register, as well as placing a reti() at the end of the interrupt routine.

Use this attribute in the attributes parameter of the ISR macro.

Note

According to GCC documentation, the only code supported in naked functions is inline assembly.

21.17.2.10 ISR_NOBLOCK `#define ISR_NOBLOCK`

ISR runs with global interrupts initially enabled. The interrupt enable flag is activated by the compiler as early as possible within the ISR to ensure minimal processing delay for nested interrupts.

This may be used to create nested ISRs, however care should be taken to avoid stack overflows, or to avoid infinitely entering the ISR for those cases where the AVR hardware does not clear the respective interrupt flag before entering the ISR.

Use this attribute in the attributes parameter of the ISR macro.

21.17.2.11 ISR_NOGCCISR `#define ISR_NOGCCISR`

Do not generate `__gcc_isr pseudo instructions` for this ISR. This has an effect with `GCC 8` and newer only.

Use this attribute in the attributes parameter of the ISR macro.

21.17.2.12 ISR_NOICF `#define ISR_NOICF`

Avoid identical-code-folding optimization against this ISR. This has an effect with GCC 5 and newer only.

Use this attribute in the attributes parameter of the ISR macro.

21.17.2.13 reti `#define reti( ) __asm__ __volatile__ ("reti" ::: "memory")`

Returns from an interrupt routine, enabling global interrupts. This should be the last command executed before leaving an ISR defined with the ISR_NAKED attribute.

This macro actually compiles into a single line of assembly, so there is no function call overhead.

Note

According to the GCC documentation, the only code supported in naked functions is inline assembly.

21.17.2.14 sei `#define sei( ) __asm__ __volatile__ ("sei" ::: "memory")`

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. However, the macro also implies a *memory barrier* which can cause additional loss of optimization.

In order to implement atomic access to multi-byte objects, consider using the macros from <util/atomic.h>, rather than implementing them manually with cli() and sei().

21.17.2.15 SIGNAL `#define SIGNAL( vector )`

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

This is the same as the ISR macro without optional attributes.

Deprecated Do not use SIGNAL() in new code. Use ISR() instead.

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

Macros

- #define _PROTECTED_WRITE(reg, value)
- #define _PROTECTED_WRITE_SPM(reg, value)

21.18.1 Detailed Description

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

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

Note that this file always includes the following files:

```
#include <avr/sfr_defs.h>
#include <avr/portpins.h>
#include <avr/common.h>
#include <avr/version.h>
```

See <avr/sfr_defs.h>: Special function registers for more details about that header file.

Included are definitions of the IO register set and their respective bit values as specified in the Atmel documentation. Note that inconsistencies in naming conventions, so even identical functions sometimes get different names on different devices.

Also included are the specific names useable for interrupt function definitions as documented here.

Finally, the following macros are defined:

- **RAMEND**
  The last on-chip RAM address.

- **XRAMEND**
  The last possible RAM location that is addressable. This is equal to RAMEND for devices that do not allow for external RAM. For devices that allow external RAM, this will be larger than RAMEND.

- **E2END**
  The last EEPROM address.

- **FLASHEND**
  The last byte address in the Flash program space.

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

- **E2PAGESIZE**
  The size of the EEPROM page.

21.18.2 Macro Definition Documentation

21.18.2.1 _PROTECTED_WRITE

```
#define _PROTECTED_WRITE(
        reg,
        value )
```

Write value `value` to IO register `reg` that is protected through the Xmega configuration change protection (CCP) mechanism. This implements the timed sequence that is required for CCP.

Example to modify the CPU clock:

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

_PROTECTED_WRITE(CLK_PSCTRL, CLK_PSADIV0_bm);
_PROTECTED_WRITE(CLK_CTRL, CLK_SCLKSEL0_bm);
```

21.18.2.2 _PROTECTED_WRITE_SPM `#define _PROTECTED_WRITE_SPM( reg, value )`

Write value `value` to register `reg` that is protected through the Xmega configuration change protection (CCP) key for self programming (SPM). This implements the timed sequence that is required for CCP.

Example to modify the CPU clock:

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

_PROTECTED_WRITE_SPM(NVMCTRL_CTRLA, NVMCTRL_CMD_PAGEERASEWRITE_gc);
```

21.19 <avr/lock.h>: Lockbit Support

Introduction

The Lockbit API allows a user to specify the lockbit settings for the specific AVR device they are compiling for. These lockbit settings will be placed in a special section in the ELF output file, after linking.

Programming tools can take advantage of the lockbit information embedded in the ELF file, by extracting this information and determining if the lockbits need to be programmed after programming the Flash and EEPROM memories. This also allows a single ELF file to contain all the information needed to program an AVR.

To use the Lockbit API, include the <avr/io.h> header file, which in turn automatically includes the individual I/O header file and the <avr/lock.h> file. These other two files provides everything necessary to set the AVR lockbits.

Lockbit API

Each I/O header file may define up to 3 macros that controls what kinds of lockbits are available to the user.

If __LOCK_BITS_EXIST is defined, then two lock bits are available to the user and 3 mode settings are defined for these two bits.

If __BOOT_LOCK_BITS_0_EXIST is defined, then the two BLB0 lock bits are available to the user and 4 mode settings are defined for these two bits.

If __BOOT_LOCK_BITS_1_EXIST is defined, then the two BLB1 lock bits are available to the user and 4 mode settings are defined for these two bits.

If __BOOT_LOCK_APPLICATION_TABLE_BITS_EXIST is defined then two lock bits are available to set the locking mode for the Application Table Section (which is used in the XMEGA family).

If __BOOT_LOCK_APPLICATION_BITS_EXIST is defined then two lock bits are available to set the locking mode for the Application Section (which is used in the XMEGA family).

If __BOOT_LOCK_BOOT_BITS_EXIST is defined then two lock bits are available to set the locking mode for the Boot Loader Section (which is used in the XMEGA family).

The AVR lockbit modes have inverted values, logical 1 for an unprogrammed (disabled) bit and logical 0 for a programmed (enabled) bit. The defined macros for each individual lock bit represent this in their definition by a bit-wise inversion of a mask. For example, the LB_MODE_3 macro is defined as:

```
    #define LB_MODE_3   (0xFC)
`
```

To combine the lockbit mode macros together to represent a whole byte, use the bitwise AND operator, like so:

```
(LB_MODE_3 & BLB0_MODE_2)
```

<avr/lock.h> also defines a macro that provides a default lockbit value: LOCKBITS_DEFAULT which is defined to be 0xFF.

See the AVR device specific datasheet for more details about these lock bits and the available mode settings.

A convenience macro, LOCKMEM, is defined as a GCC attribute for a custom-named section of ".lock".

A convenience macro, LOCKBITS, is defined that declares a variable, __lock, of type unsigned char with the attribute defined by LOCKMEM. This variable allows the end user to easily set the lockbit data.

Note

If a device-specific I/O header file has previously defined LOCKMEM, then LOCKMEM is not redefined. If a device-specific I/O header file has previously defined LOCKBITS, then LOCKBITS is not redefined. LOCKBITS is currently known to be defined in the I/O header files for the XMEGA devices.

API Usage Example

Putting all of this together is easy:

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

LOCKBITS = (LB_MODE_1 & BLB0_MODE_3 & BLB1_MODE_4);

int main(void)
{
    return 0;
}
```

Or:

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

unsigned char __lock __attribute__((section (".lock"))) =
    (LB_MODE_1 & BLB0_MODE_3 & BLB1_MODE_4);

int main(void)
{
    return 0;
}
```

However there are a number of caveats that you need to be aware of to use this API properly.

Be sure to include <avr/io.h> to get all of the definitions for the API. The LOCKBITS macro defines a global variable to store the lockbit data. This variable is assigned to its own linker section. Assign the desired lockbit values immediately in the variable initialization.

The .lock section in the ELF file will get its values from the initial variable assignment ONLY. This means that you can NOT assign values to this variable in functions and the new values will not be put into the ELF .lock section.

The global variable is declared in the LOCKBITS macro has two leading underscores, which means that it is reserved for the "implementation", meaning the library, so it will not conflict with a user-named variable.

You must initialize the lockbit variable to some meaningful value, even if it is the default value. This is because the lockbits default to a logical 1, meaning unprogrammed. Normal uninitialized data defaults to all locgial zeros. So it is vital that all lockbits are initialized, even with default data. If they are not, then the lockbits may not programmed to the desired settings and can possibly put your device into an unrecoverable state.

Be sure to have the -mmcu=*device* flag in your compile command line and your linker command line to have the correct device selected and to have the correct I/O header file included when you include <avr/io.h>.

You can print out the contents of the .lock section in the ELF file by using this command line:

```
avr-objdump -s -j .lock <ELF file>
```

21.20 <avr/pgmspace.h>: Program Space Utilities

Macros

- #define PROGMEM_FAR __attribute__((__section__(".progmemx.data")))
- #define PROGMEM __attribute__((__progmem__))
- #define PSTR(str) ({ static const PROGMEM char c[ ] = (str); &c[0]; })
- #define PSTR_FAR(str) ({ static const PROGMEM_FAR char c[ ] = (str); pgm_get_far_address(c[0]); })
- #define pgm_read_byte_near(__addr) __LPM ((uint16_t)(__addr))
- #define pgm_read_word_near(__addr) __LPM_word ((uint16_t)(__addr))
- #define pgm_read_dword_near(__addr) __LPM_dword ((uint16_t)(__addr))
- #define pgm_read_qword_near(__addr) __LPM_qword ((uint16_t)(__addr))
- #define pgm_read_float_near(addr) pgm_read_float (addr)
- #define pgm_read_ptr_near(__addr) ((void*) __LPM_word ((uint16_t)(__addr)))
- #define pgm_read_byte_far(__addr) __ELPM (__addr)
- #define pgm_read_word_far(__addr) __ELPM_word (__addr)
- #define pgm_read_dword_far(__addr) __ELPM_dword (__addr)
- #define pgm_read_qword_far(__addr) __ELPM_qword (__addr)
- #define pgm_read_ptr_far(__addr) ((void*) __ELPM_word (__addr))
- #define pgm_read_byte(__addr) pgm_read_byte_near(__addr)
- #define pgm_read_word(__addr) pgm_read_word_near(__addr)
- #define pgm_read_dword(__addr) pgm_read_dword_near(__addr)
- #define pgm_read_qword(__addr) pgm_read_qword_near(__addr)
- #define pgm_read_ptr(__addr) pgm_read_ptr_near(__addr)
- #define pgm_get_far_address(var)

Functions

- static char pgm_read_char (const char *__addr)
- static unsigned char pgm_read_unsigned_char (const unsigned char *__addr)
- static signed char pgm_read_signed_char (const signed char *__addr)
- static uint8_t pgm_read_u8 (const uint8_t *__addr)
- static int8_t pgm_read_i8 (const int8_t *__addr)
- static short pgm_read_short (const short *__addr)
- static unsigned short pgm_read_unsigned_short (const unsigned short *__addr)
- static uint16_t pgm_read_u16 (const uint16_t *__addr)
- static int16_t pgm_read_i16 (const int16_t *__addr)
- static int pgm_read_int (const int *__addr)
- static signed pgm_read_signed (const signed *__addr)
- static unsigned pgm_read_unsigned (const unsigned *__addr)
- static signed int pgm_read_signed_int (const signed int *__addr)
- static unsigned int pgm_read_unsigned_int (const unsigned int *__addr)
- static __int24 pgm_read_i24 (const __int24 *__addr)
- static __uint24 pgm_read_u24 (const __uint24 *__addr)
- static uint32_t pgm_read_u32 (const uint32_t *__addr)
- static int32_t pgm_read_i32 (const int32_t *__addr)
- static long pgm_read_long (const long *__addr)
- static unsigned long pgm_read_unsigned_long (const unsigned long *__addr)
- static long long pgm_read_long_long (const long long *__addr)
- static unsigned long long pgm_read_unsigned_long_long (const unsigned long long *__addr)
- static uint64_t pgm_read_u64 (const uint64_t *__addr)
- static int64_t pgm_read_i64 (const int64_t *__addr)
- static float pgm_read_float (const float *__addr)

- static double pgm_read_double (const double *__addr)
- static long double pgm_read_long_double (const long double *__addr)
- static char pgm_read_char_far (uint_farptr_t __addr)
- static unsigned char pgm_read_unsigned_char_far (uint_farptr_t __addr)
- static signed char pgm_read_signed_char_far (uint_farptr_t __addr)
- static uint8_t pgm_read_u8_far (uint_farptr_t __addr)
- static int8_t pgm_read_i8_far (uint_farptr_t __addr)
- static short pgm_read_short_far (uint_farptr_t __addr)
- static unsigned short pgm_read_unsigned_short_far (uint_farptr_t __addr)
- static uint16_t pgm_read_u16_far (uint_farptr_t __addr)
- static int16_t pgm_read_i16_far (uint_farptr_t __addr)
- static int pgm_read_int_far (uint_farptr_t __addr)
- static unsigned pgm_read_unsigned_far (uint_farptr_t __addr)
- static unsigned int pgm_read_unsigned_int_far (uint_farptr_t __addr)
- static signed pgm_read_signed_far (uint_farptr_t __addr)
- static signed int pgm_read_signed_int_far (uint_farptr_t __addr)
- static long pgm_read_long_far (uint_farptr_t __addr)
- static unsigned long pgm_read_unsigned_long_far (uint_farptr_t __addr)
- static __int24 pgm_read_i24_far (uint_farptr_t __addr)
- static __uint24 pgm_read_u24_far (uint_farptr_t __addr)
- static uint32_t pgm_read_u32_far (uint_farptr_t __addr)
- static int32_t pgm_read_i32_far (uint_farptr_t __addr)
- static long long pgm_read_long_long_far (uint_farptr_t __addr)
- static unsigned long long pgm_read_unsigned_long_long_far (uint_farptr_t __addr)
- static uint64_t pgm_read_u64_far (uint_farptr_t __addr)
- static int64_t pgm_read_i64_far (uint_farptr_t __addr)
- static float pgm_read_float_far (uint_farptr_t __addr)
- static double pgm_read_double_far (uint_farptr_t __addr)
- static long double pgm_read_long_double_far (uint_farptr_t __addr)
- const void * memchr_P (const void *, int __val, size_t __len)
- int memcmp_P (const void *, const void *, size_t)
- void * memccpy_P (void *, const void *, int __val, size_t)
- void * memcpy_P (void *, const void *, size_t)
- void * memmem_P (const void *, size_t, const void *, size_t)
- const void * memrchr_P (const void *, int __val, size_t __len)
- char * strcat_P (char *, const char *)
- const char * strchr_P (const char *, int __val)
- const char * strchrnul_P (const char *, int __val)
- int strcmp_P (const char *, const char *)
- char * strcpy_P (char *, const char *)
- int strcasecmp_P (const char *, const char *)
- char * strcasestr_P (const char *, const char *)
- size_t strcspn_P (const char *__s, const char *__reject)
- size_t strlcat_P (char *, const char *, size_t)
- size_t strlcpy_P (char *, const char *, size_t)
- size_t strnlen_P (const char *, size_t)
- int strncmp_P (const char *, const char *, size_t)
- int strncasecmp_P (const char *, const char *, size_t)
- char * strncat_P (char *, const char *, size_t)
- char * strncpy_P (char *, const char *, size_t)
- char * strpbrk_P (const char *__s, const char *__accept)
- const char * strrchr_P (const char *, int __val)
- char * strsep_P (char **__sp, const char *__delim)
- size_t strspn_P (const char *__s, const char *__accept)
- char * strstr_P (const char *, const char *)

- char * strtok_P (char *__s, const char *__delim)
- char * strtok_rP (char *__s, const char *__delim, char **__last)
- size_t strlen_PF (uint_farptr_t src)
- size_t strnlen_PF (uint_farptr_t src, size_t len)
- void * memcpy_PF (void *dest, uint_farptr_t src, size_t len)
- char * strcpy_PF (char *dest, uint_farptr_t src)
- char * strncpy_PF (char *dest, uint_farptr_t src, size_t len)
- char * strcat_PF (char *dest, uint_farptr_t src)
- size_t strlcat_PF (char *dst, uint_farptr_t src, size_t siz)
- char * strncat_PF (char *dest, uint_farptr_t src, size_t len)
- int strcmp_PF (const char *s1, uint_farptr_t s2)
- int strncmp_PF (const char *s1, uint_farptr_t s2, size_t n)
- int strcasecmp_PF (const char *s1, uint_farptr_t s2)
- int strncasecmp_PF (const char *s1, uint_farptr_t s2, size_t n)
- uint_farptr_t strchr_PF (uint_farptr_t, int __val)
- char * strstr_PF (const char *s1, uint_farptr_t s2)
- size_t strlcpy_PF (char *dst, uint_farptr_t src, size_t siz)
- int memcmp_PF (const void *, uint_farptr_t, size_t)
- static size_t strlen_P (const char *s)
- template<typename T >
  T pgm_read< T > (const T *addr)
- template<typename T >
  T pgm_read_far< T > (uint_farptr_t addr)

21.20.1 Detailed Description

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

The functions in this module provide interfaces for a program to access data stored in program space (flash memory) of the device.

Note

These functions are an attempt to provide some compatibility with header files that come with IAR C, to make porting applications between different compilers easier. This is not 100% compatibility though (GCC does not have full support for multiple address spaces yet).

If you are working with strings which are completely based in RAM, use the standard string functions described in <string.h>: Strings.

If possible, put your constant tables in the lower 64 KB and use pgm_read_byte_near() or pgm_read_word_near() instead of pgm_read_byte_far() or pgm_read_word_far() since it is more efficient that way, and you can still use the upper 64K for executable code. All functions that are suffixed with a `_P` *require* their arguments to be in the lower 64 KB of the flash ROM, as they do not use ELPM instructions. This is normally not a big concern as the linker setup arranges any program space constants declared using the macros from this header file so they are placed right after the interrupt vectors, and in front of any executable code. However, it can become a problem if there are too many of these constants, or for bootloaders on devices with more than 64 KB of ROM. *All these functions will not work in that situation.*

For **Xmega** devices, make sure the NVM controller command register (`NVM.CMD` or `NVM_CMD`) is set to 0x00 (NOP) before using any of these functions.

21.20.2 Macro Definition Documentation

21.20.2.1 pgm_get_far_address `#define pgm_get_far_address( var )`

Value:
```
(({                                                     \
    uint_farptr_t __tmp;                                \
                                                        \
    __asm__ __volatile__ (                              \
        "ldi    %A0, lo8(%1)"           "\n\t"          \
        "ldi    %B0, hi8(%1)"           "\n\t"          \
        "ldi    %C0, hh8(%1)"           "\n\t"          \
        "clr    %D0"                                    \
        :   "=d" (__tmp)                                \
        :   "i"  (&(var))                               \
    );                                                  \
    __tmp;                                              \
}))
```

This macro evaluates to a uint_farptr_t 32-bit "far" pointer (only 24 bits used) to data even beyond the 64 KiB limit for the 16-bit ordinary pointer. It is similar to the '&' operator, with some limitations. Example:

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

// Section .progmemx.data is located after all the code sections.
__attribute__((section(".progmemx.data")))
const int data[] = { 2, 3, 5, 7, 9, 11 };

int get_data (uint8_t idx)
{
    uint_farptr_t pdata = pgm_get_far_address (data[0]);
    return pgm_read_int_far (pdata + idx * sizeof(int));
}
```

Comments:

- The overhead is minimal and it's mainly due to the 32-bit size operation.
- 24 bit sizes guarantees the code compatibility for use in future devices.
- `var` has to be resolved at link-time as an existing symbol, i.e. a simple variable name, an array name, or an array or structure element provided the offset is known at compile-time, and `var` is located in static storage, etc.
- The returned value is the symbol's VMA (virtual memory address) determined by the linker and falls in the corresponding memory region. The AVR Harvard architecture requires non-overlapping VMA areas for the multiple memory regions in the processor: Flash ROM, RAM, and EEPROM. Typical offset for these are `0x0`, `0x800xx0`, and `0x810000` respectively, derived from the linker script used and linker options.

21.20.2.2 pgm_read_byte `#define pgm_read_byte( __addr ) pgm_read_byte_near(__addr)`

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

21.20.2.3 pgm_read_byte_far `#define pgm_read_byte_far( __addr ) __ELPM (__addr)`

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

21.20.2.4 pgm_read_byte_near `#define pgm_read_byte_near( __addr ) __LPM ((uint16_t)(__addr))`

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

21.20.2.5 pgm_read_dword `#define pgm_read_dword( __addr ) pgm_read_dword_near(__addr)`

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

21.20.2.6 pgm_read_dword_far `#define pgm_read_dword_far( __addr ) __ELPM_dword (__addr)`

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

21.20.2.7 pgm_read_dword_near `#define pgm_read_dword_near( __addr ) __LPM_dword ((uint16_t)(__addr))`

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

21.20.2.8 pgm_read_float_near `#define pgm_read_float_near( addr ) pgm_read_float (addr)`

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

21.20.2.9 pgm_read_ptr `#define pgm_read_ptr( __addr ) pgm_read_ptr_near(__addr)`

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

21.20.2.10 pgm_read_ptr_far `#define pgm_read_ptr_far( __addr ) ((void*) __ELPM_word (__addr))`

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

21.20.2.11 pgm_read_ptr_near `#define pgm_read_ptr_near( __addr ) ((void*) __LPM_word ((uint16_t)(__addr)))`

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

21.20.2.12 pgm_read_qword `#define pgm_read_qword( __addr ) pgm_read_qword_near(__addr)`

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

21.20.2.13 pgm_read_qword_far `#define pgm_read_qword_far( __addr ) __ELPM_qword (__addr)`

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

21.20.2.14 pgm_read_qword_near `#define pgm_read_qword_near( __addr ) __LPM_qword ((uint16_t)(__addr))`

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

21.20.2.15 pgm_read_word

```
#define pgm_read_word( __addr ) pgm_read_word_near(__addr)
```

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

21.20.2.16 pgm_read_word_far

```
#define pgm_read_word_far( __addr ) __ELPM_word (__addr)
```

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

21.20.2.17 pgm_read_word_near

```
#define pgm_read_word_near( __addr ) __LPM_word ((uint16_t)(__addr))
```

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

21.20.2.18 PROGMEM

```
#define PROGMEM __attribute__((__progmem__))
```

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

21.20.2.19 PROGMEM_FAR

```
#define PROGMEM_FAR __attribute__((__section__(".progmemx.data")))
```

Attribute to use in order to declare an object being located in far flash ROM. This is similar to PROGMEM, except that it puts the static storage object in section `.progmemx.data`. In order to access the object, the `pgm_read_*_far` and `_PF` functions declare in this header can be used. In order to get its address, see pgm_get_far_address().

It only makes sense to put read-only objects in this section, though the compiler does not diagnose when this is not the case.

21.20.2.20 PSTR

```
#define PSTR( str ) ({ static const PROGMEM char c[] = (str); &c[0]; })
```

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

21.20.2.21 PSTR_FAR

```
#define PSTR_FAR( str ) ({ static const PROGMEM_FAR char c[] = (str); pgm_get_far_address(c[0]);
})
```

Used to define a string literal in far program space, and to return its address of type uint_farptr_t.

21.20.3 Function Documentation

21.20.3.1 memccpy_P()

```
void * memccpy_P (
    void * dest,
    const void * src,
    int val,
    size_t len )
```

This function is similar to memccpy() except that `src` is pointer to a string in program space.

21.20.3.2 memchr_P()

```
const void * memchr_P (
    const void * s,
    int val,
    size_t len )
```

Scan flash memory for a character.

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

Returns

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

21.20.3.3 memcmp_P()

```
int memcmp_P (
    const void * s1,
    const void * s2,
    size_t len )
```

Compare memory areas.

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

Returns

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

21.20.3.4 memcmp_PF()

```
int memcmp_PF (
    const void * s1,
    uint_farptr_t s2,
    size_t len )
```

Compare memory areas.

The memcmp_PF() function compares the first `len` bytes of the memory areas `s1` and flash `s2`. The comparision is performed using unsigned char operations. It is an equivalent of memcmp_P() function, except that it is capable working on all FLASH including the exteded area above 64kB.

Returns

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

21.20.3.5 memcpy_P()

```
void * memcpy_P (
    void * dest,
    const void * src,
    size_t n )
```

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

Returns

The memcpy_P() function returns a pointer to dest.

21.20.3.6 memcpy_PF()

```
void * memcpy_PF (
    void * dest,
    uint_farptr_t src,
    size_t n )
```

Copy a memory block from flash to SRAM.

The memcpy_PF() function is similar to memcpy(), except the data is copied from the program space and is addressed using a far pointer.

Parameters

dest	A pointer to the destination buffer
src	A far pointer to the origin of data in flash memory
n	The number of bytes to be copied

Returns

The memcpy_PF() function returns a pointer to *dst*. The contents of RAMPZ SFR are undefined when the function returns.

21.20.3.7 memmem_P()

```
void * memmem_P (
    const void * s1,
    size_t len1,
    const void * s2,
    size_t len2 )
```

The memmem_P() function is similar to memmem() except that `s2` is pointer to a string in program space.

21.20.3.8 memrchr_P()

```
const void memrchr_P (
    const void * src,
    int val,
    size_t len )
```

The memrchr_P() function is like the memchr_P() function, except that it searches backwards from the end of the `len` bytes pointed to by `src` instead of forwards from the front. (Glibc, GNU extension.)

Returns

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

21.20.3.9 pgm_read< T >()

template<typename T >
T pgm_read< T > (
const T * *addr*)

Read an object of type T from program memory address addr and return it. This template is only available when macro __pgm_read_template__ is defined.

21.20.3.10 pgm_read_char()

char pgm_read_char (
const char * *__addr*) [inline], [static]

Read a char from 16-bit (near) byte-address __addr. The address is in the lower 64 KiB of program memory.

21.20.3.11 pgm_read_char_far()

char pgm_read_char_far (
uint_farptr_t *__addr*) [inline], [static]

Read a char from far byte-address __addr. The address is in the program memory.

21.20.3.12 pgm_read_double()

double pgm_read_double (
const double * *__addr*) [inline], [static]

Read a double from 16-bit (near) byte-address __addr. The address is in the lower 64 KiB of program memory.

21.20.3.13 pgm_read_double_far()

double pgm_read_double_far (
uint_farptr_t *__addr*) [inline], [static]

Read a double from far byte-address __addr. The address is in the program memory.

21.20.3.14 pgm_read_far< T >()

template<typename T >
T pgm_read_far< T > (
uint_farptr_t *addr*)

Read an object of type T from program memory address addr and return it. This template is only available when macro __pgm_read_template__ is defined.

21.20.3.15 pgm_read_float()

float pgm_read_float (
const float * *__addr*) [inline], [static]

Read a float from 16-bit (near) byte-address __addr. The address is in the lower 64 KiB of program memory.

21.20.3.16 pgm_read_float_far() float pgm_read_float_far (uint_farptr_t __addr) [inline], [static]

Read a float from far byte-address __addr. The address is in the program memory.

21.20.3.17 pgm_read_i16() int16_t pgm_read_i16 (const int16_t * __addr) [inline], [static]

Read an int16_t from 16-bit (near) byte-address __addr. The address is in the lower 64 KiB of program memory.

21.20.3.18 pgm_read_i16_far() int16_t pgm_read_i16_far (uint_farptr_t __addr) [inline], [static]

Read an int16_t from far byte-address __addr. The address is in the program memory.

21.20.3.19 pgm_read_i24() __int24 pgm_read_i24 (const __int24 * __addr) [inline], [static]

Read an __int24 from 16-bit (near) byte-address __addr. The address is in the lower 64 KiB of program memory.

21.20.3.20 pgm_read_i24_far() __int24 pgm_read_i24_far (uint_farptr_t __addr) [inline], [static]

Read an __int24 from far byte-address __addr. The address is in the program memory.

21.20.3.21 pgm_read_i32() int32_t pgm_read_i32 (const int32_t * __addr) [inline], [static]

Read an int32_t from 16-bit (near) byte-address __addr. The address is in the lower 64 KiB of program memory.

21.20.3.22 pgm_read_i32_far() int32_t pgm_read_i32_far (uint_farptr_t __addr) [inline], [static]

Read an int32_t from far byte-address __addr. The address is in the program memory.

21.20.3.23 pgm_read_i64() int64_t pgm_read_i64 (const int64_t * __addr) [inline], [static]

Read an int64_t from 16-bit (near) byte-address __addr. The address is in the lower 64 KiB of program memory.

21.20.3.24 pgm_read_i64_far() int64_t pgm_read_i64_far (uint_farptr_t __addr) [inline], [static]

Read an int64_t from far byte-address __addr. The address is in the program memory.

21.20.3.25 pgm_read_i8()

int8_t pgm_read_i8 (
const int8_t * __addr) [inline], [static]

Read an int8_t from 16-bit (near) byte-address __addr. The address is in the lower 64 KiB of program memory.

21.20.3.26 pgm_read_i8_far()

int8_t pgm_read_i8_far (
uint_farptr_t __addr) [inline], [static]

Read an int8_t from far byte-address __addr. The address is in the program memory.

21.20.3.27 pgm_read_int()

int pgm_read_int (
const int * __addr) [inline], [static]

Read an int from 16-bit (near) byte-address __addr. The address is in the lower 64 KiB of program memory.

21.20.3.28 pgm_read_int_far()

int pgm_read_int_far (
uint_farptr_t __addr) [inline], [static]

Read an int from far byte-address __addr. The address is in the program memory.

21.20.3.29 pgm_read_long()

long pgm_read_long (
const long * __addr) [inline], [static]

Read a long from 16-bit (near) byte-address __addr. The address is in the lower 64 KiB of program memory.

21.20.3.30 pgm_read_long_double()

long double pgm_read_long_double (
const long double * __addr) [inline], [static]

Read a long double from 16-bit (near) byte-address __addr. The address is in the lower 64 KiB of program memory.

21.20.3.31 pgm_read_long_double_far()

long double pgm_read_long_double_far (
uint_farptr_t __addr) [inline], [static]

Read a long double from far byte-address __addr. The address is in the program memory.

21.20.3.32 pgm_read_long_far()

long pgm_read_long_far (
uint_farptr_t __addr) [inline], [static]

Read a long from far byte-address __addr. The address is in the program memory.

21.20.3.33 pgm_read_long_long()

long long pgm_read_long_long (
const long long * __addr) [inline], [static]

Read a long long from 16-bit (near) byte-address __addr. The address is in the lower 64 KiB of program memory.

21.20.3.34 pgm_read_long_long_far()

long long pgm_read_long_long_far (uint_farptr_t __addr) [inline], [static]

Read a long long from far byte-address __addr. The address is in the program memory.

21.20.3.35 pgm_read_short()

short pgm_read_short (const short * __addr) [inline], [static]

Read a short from 16-bit (near) byte-address __addr. The address is in the lower 64 KiB of program memory.

21.20.3.36 pgm_read_short_far()

short pgm_read_short_far (uint_farptr_t __addr) [inline], [static]

Read a short from far byte-address __addr. The address is in the program memory.

21.20.3.37 pgm_read_signed()

signed pgm_read_signed (const signed * __addr) [inline], [static]

Read a signed from 16-bit (near) byte-address __addr. The address is in the lower 64 KiB of program memory.

21.20.3.38 pgm_read_signed_char()

signed char pgm_read_signed_char (const signed char * __addr) [inline], [static]

Read a signed char from 16-bit (near) byte-address __addr. The address is in the lower 64 KiB of program memory.

21.20.3.39 pgm_read_signed_char_far()

signed char pgm_read_signed_char_far (uint_farptr_t __addr) [inline], [static]

Read a signed char from far byte-address __addr. The address is in the program memory.

21.20.3.40 pgm_read_signed_far()

signed pgm_read_signed_far (uint_farptr_t __addr) [inline], [static]

Read a signed from far byte-address __addr. The address is in the program memory.

21.20.3.41 pgm_read_signed_int()

signed int pgm_read_signed_int (const signed int * __addr) [inline], [static]

Read a signed int from 16-bit (near) byte-address __addr. The address is in the lower 64 KiB of program memory.

21.20.3.42 pgm_read_signed_int_far()

signed int pgm_read_signed_int_far (uint_farptr_t __addr) [inline], [static]

Read a signed int from far byte-address __addr. The address is in the program memory.

21.20.3.43 pgm_read_u16()

```
uint16_t pgm_read_u16 (
        const uint16_t * __addr ) [inline], [static]
```

Read an uint16_t from 16-bit (near) byte-address __addr. The address is in the lower 64 KiB of program memory.

21.20.3.44 pgm_read_u16_far()

```
uint16_t pgm_read_u16_far (
        uint_farptr_t __addr ) [inline], [static]
```

Read an uint16_t from far byte-address __addr. The address is in the program memory.

21.20.3.45 pgm_read_u24()

```
__uint24 pgm_read_u24 (
        const __uint24 * __addr ) [inline], [static]
```

Read an __uint24 from 16-bit (near) byte-address __addr. The address is in the lower 64 KiB of program memory.

21.20.3.46 pgm_read_u24_far()

```
__uint24 pgm_read_u24_far (
        uint_farptr_t __addr ) [inline], [static]
```

Read an __uint24 from far byte-address __addr. The address is in the program memory.

21.20.3.47 pgm_read_u32()

```
uint32_t pgm_read_u32 (
        const uint32_t * __addr ) [inline], [static]
```

Read an uint32_t from 16-bit (near) byte-address __addr. The address is in the lower 64 KiB of program memory.

21.20.3.48 pgm_read_u32_far()

```
uint32_t pgm_read_u32_far (
        uint_farptr_t __addr ) [inline], [static]
```

Read an uint32_t from far byte-address __addr. The address is in the program memory.

21.20.3.49 pgm_read_u64()

```
uint64_t pgm_read_u64 (
        const uint64_t * __addr ) [inline], [static]
```

Read an uint64_t from 16-bit (near) byte-address __addr. The address is in the lower 64 KiB of program memory.

21.20.3.50 pgm_read_u64_far()

```
uint64_t pgm_read_u64_far (
        uint_farptr_t __addr ) [inline], [static]
```

Read an uint64_t from far byte-address __addr. The address is in the program memory.

21.20.3.51 pgm_read_u8()

```
uint8_t pgm_read_u8 (
        const uint8_t * __addr ) [inline], [static]
```

Read an uint8_t from 16-bit (near) byte-address __addr. The address is in the lower 64 KiB of program memory.

21.20.3.52 pgm_read_u8_far() uint8_t pgm_read_u8_far (uint_farptr_t __addr) [inline], [static]

Read an uint8_t from far byte-address __addr. The address is in the program memory.

21.20.3.53 pgm_read_unsigned() unsigned pgm_read_unsigned (const unsigned * __addr) [inline], [static]

Read an unsigned from 16-bit (near) byte-address __addr. The address is in the lower 64 KiB of program memory.

21.20.3.54 pgm_read_unsigned_char() unsigned char pgm_read_unsigned_char (const unsigned char * __addr) [inline], [static]

Read an unsigned char from 16-bit (near) byte-address __addr. The address is in the lower 64 KiB of program memory.

21.20.3.55 pgm_read_unsigned_char_far() unsigned char pgm_read_unsigned_char_far (uint_farptr_t __addr) [inline], [static]

Read an unsigned char from far byte-address __addr. The address is in the program memory.

21.20.3.56 pgm_read_unsigned_far() unsigned pgm_read_unsigned_far (uint_farptr_t __addr) [inline], [static]

Read an unsigned from far byte-address __addr. The address is in the program memory.

21.20.3.57 pgm_read_unsigned_int() unsigned int pgm_read_unsigned_int (const unsigned int * __addr) [inline], [static]

Read an unsigned int from 16-bit (near) byte-address __addr. The address is in the lower 64 KiB of program memory.

21.20.3.58 pgm_read_unsigned_int_far() unsigned int pgm_read_unsigned_int_far (uint_farptr_t __addr) [inline], [static]

Read an unsigned int from far byte-address __addr. The address is in the program memory.

21.20.3.59 pgm_read_unsigned_long() unsigned long pgm_read_unsigned_long (const unsigned long * __addr) [inline], [static]

Read an unsigned long from 16-bit (near) byte-address __addr. The address is in the lower 64 KiB of program memory.

21.20.3.60 pgm_read_unsigned_long_far() unsigned long pgm_read_unsigned_long_far (uint_farptr_t __addr) [inline], [static]

Read an unsigned long from far byte-address __addr. The address is in the program memory.

21.20.3.61 pgm_read_unsigned_long_long() unsigned long long pgm_read_unsigned_long_long (const unsigned long long * __addr) [inline], [static]

Read an unsigned long long from 16-bit (near) byte-address __addr. The address is in the lower 64 KiB of program memory.

21.20.3.62 pgm_read_unsigned_long_long_far() unsigned long long pgm_read_unsigned_long_long_↩ far (uint_farptr_t __addr) [inline], [static]

Read an unsigned long long from far byte-address __addr. The address is in the program memory.

21.20.3.63 pgm_read_unsigned_short() unsigned short pgm_read_unsigned_short (const unsigned short * __addr) [inline], [static]

Read an unsigned short from 16-bit (near) byte-address __addr. The address is in the lower 64 KiB of program memory.

21.20.3.64 pgm_read_unsigned_short_far() unsigned short pgm_read_unsigned_short_far (uint_farptr_t __addr) [inline], [static]

Read an unsigned short from far byte-address __addr. The address is in the program memory.

21.20.3.65 strcasecmp_P() int strcasecmp_P (const char * s1, const char * s2)

Compare two strings ignoring case.

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

Parameters

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

Returns

The strcasecmp_P() function returns an integer less than, equal to, or greater than zero if s1 is found, respectively, to be less than, to match, or be greater than s2. A consequence of the ordering used by strcasecmp_P() is that if s1 is an initial substring of s2, then s1 is considered to be "less than" s2.

21.20.3.66 strcasecmp_PF() int strcasecmp_PF (const char * s1, uint_farptr_t s2)

Compare two strings ignoring case.

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

Parameters

s1	A pointer to the first string in SRAM
s2	A far pointer to the second string in Flash

Returns

The strcasecmp_PF() function returns an integer less than, equal to, or greater than zero if *s1* is found, respectively, to be less than, to match, or be greater than *s2*. The contents of RAMPZ SFR are undefined when the function returns.

21.20.3.67 strcasestr_P()

```
char * strcasestr_P (
    const char * s1,
    const char * s2 )
```

This funtion is similar to strcasestr() except that `s2` is pointer to a string in program space.

21.20.3.68 strcat_P()

```
char * strcat_P (
    char * dest,
    const char * src )
```

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

Returns

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

21.20.3.69 strcat_PF()

```
char * strcat_PF (
    char * dst,
    uint_farptr_t src )
```

Concatenates two strings.

The strcat_PF() function is similar to strcat() except that the *src* string must be located in program space (flash) and is addressed using a far pointer

Parameters

dst	A pointer to the destination string in SRAM
src	A far pointer to the string to be appended in Flash

Returns

The strcat_PF() function returns a pointer to the resulting string *dst*. The contents of RAMPZ SFR are undefined when the function returns

21.20.3.70 strchr_P()

```
const char * strchr_P (
    const char * s,
    int val )
```

Locate character in program space string.

The strchr_P() function locates the first occurrence of `val` (converted to a char) in the string pointed to by `s` in program space. The terminating null character is considered to be part of the string.

The strchr_P() function is similar to strchr() except that `s` is pointer to a string in program space.

Returns

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

21.20.3.71 strchr_PF()

```
uint_farptr_t strchr_PF (
    uint_farptr_t s,
    int val )
```

Locate character in far program space string.

The strchr_PF() function locates the first occurrence of `val` (converted to a char) in the string pointed to by `s` in far program space. The terminating null character is considered to be part of the string.

The strchr_PF() function is similar to strchr() except that `s` is a far pointer to a string in program space that's *not required* to be located in the lower 64 KiB block like it is the case for strchr_P().

Returns

The strchr_PF() function returns a far pointer to the matched character or `0` if the character is not found.

21.20.3.72 strchrnul_P()

```
const char * strchrnul_P (
    const char * s,
    int c )
```

The strchrnul_P() function is like strchr_P() except that if `c` is not found in `s`, then it returns a pointer to the null byte at the end of `s`, rather than `NULL`. (Glibc, GNU extension.)

Returns

The strchrnul_P() function returns a pointer to the matched character, or a pointer to the null byte at the end of `s` (i.e., `s+strlen`(s)) if the character is not found.

21.20.3.73 strcmp_P()

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

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

Returns

The strcmp_P() function returns an integer less than, equal to, or greater than zero if `s1` is found, respectively, to be less than, to match, or be greater than `s2`. A consequence of the ordering used by strcmp_P() is that if `s1` is an initial substring of `s2`, then `s1` is considered to be "less than" `s2`.

21.20.3.74 strcmp_PF()

```
int strcmp_PF (
    const char * s1,
    uint_farptr_t s2 )
```

Compares two strings.

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

Parameters

s1	A pointer to the first string in SRAM
s2	A far pointer to the second string in Flash

Returns

The strcmp_PF() function returns an integer less than, equal to, or greater than zero if *s1* is found, respectively, to be less than, to match, or be greater than *s2*. The contents of RAMPZ SFR are undefined when the function returns.

21.20.3.75 strcpy_P()

```
char * strcpy_P (
    char * dest,
    const char * src )
```

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

Returns

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

21.20.3.76 strcpy_PF()

```
char * strcpy_PF (
    char * dst,
    uint_farptr_t src )
```

Duplicate a string.

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

Parameters

dst	A pointer to the destination string in SRAM
src	A far pointer to the source string in Flash

Returns

The strcpy_PF() function returns a pointer to the destination string *dst*. The contents of RAMPZ SFR are undefined when the funcion returns.

21.20.3.77 strcspn_P()

```
size_t strcspn_P (
    const char * s,
    const char * reject )
```

The strcspn_P() function calculates the length of the initial segment of `s` which consists entirely of characters not in `reject`. This function is similar to strcspn() except that `reject` is a pointer to a string in program space.

Returns

The strcspn_P() function returns the number of characters in the initial segment of `s` which are not in the string `reject`. The terminating zero is not considered as a part of string.

21.20.3.78 strlcat_P()

```
size_t strlcat_P (
    char * dst,
    const char * src,
    size_t siz )
```

Concatenate two strings.

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

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

Returns

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

21.20.3.79 strlcat_PF()

```
size_t strlcat_PF (
    char * dst,
    uint_farptr_t src,
    size_t n )
```

Concatenate two strings.

The strlcat_PF() function is similar to strlcat(), except that the *src* string must be located in program space (flash) and is addressed using a far pointer.

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

Parameters

dst	A pointer to the destination string in SRAM
src	A far pointer to the source string in Flash
n	The total number of bytes allocated to the destination string

Returns

The strlcat_PF() function returns strlen(*src*) + MIN(*n*, strlen(initial *dst*)). If retval >= *n*, truncation occurred. The contents of RAMPZ SFR are undefined when the funcion returns.

21.20.3.80 strlcpy_P()

```
size_t strlcpy_P (
    char * dst,
    const char * src,
    size_t siz )
```

Copy a string from progmem to RAM.

Copy `src` to string `dst` of size `siz`. At most `siz-1` characters will be copied. Always NULL terminates (unless `siz` == 0). The strlcpy_P() function is similar to strlcpy() except that the `src` is pointer to a string in memory space.

Returns

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

21.20.3.81 strlcpy_PF()

```
size_t strlcpy_PF (
    char * dst,
    uint_farptr_t src,
    size_t siz )
```

Copy a string from progmem to RAM.

Copy src to string dst of size siz. At most siz-1 characters will be copied. Always NULL terminates (unless siz == 0).

Returns

The strlcpy_PF() function returns strlen(src). If retval >= siz, truncation occurred. The contents of RAMPZ SFR are undefined when the function returns.

21.20.3.82 strlen_P()

```
size_t strlen_P (
const char * src ) [inline], [static]
```

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

Returns

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

Note

strlen_P() is implemented as an inline function in the avr/pgmspace.h header file, which will check if the length of the string is a constant and known at compile time. If it is not known at compile time, the macro will issue a call to __strlen_P() which will then calculate the length of the string as normal.

21.20.3.83 strlen_PF()

```
size_t strlen_PF (
uint_farptr_t s )
```

Obtain the length of a string.

The strlen_PF() function is similar to strlen(), except that *s* is a far pointer to a string in program space.

Parameters

s	A far pointer to the string in flash

Returns

The strlen_PF() function returns the number of characters in *s*. The contents of RAMPZ SFR are undefined when the function returns.

21.20.3.84 strncasecmp_P()

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

Compare two strings ignoring case.

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

Parameters

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

Returns

The strncasecmp_P() function returns an integer less than, equal to, or greater than zero if `s1` (or the first `n` bytes thereof) is found, respectively, to be less than, to match, or be greater than `s2`. A consequence of the ordering used by strncasecmp_P() is that if `s1` is an initial substring of `s2`, then `s1` is considered to be "less than" `s2`.

21.20.3.85 strncasecmp_PF()

```
int strncasecmp_PF (
    const char * s1,
    uint_farptr_t s2,
    size_t n )
```

Compare two strings ignoring case.

The strncasecmp_PF() function is similar to strcasecmp_PF(), except it only compares the first *n* characters of *s1* and the string in flash is addressed using a far pointer.

Parameters

s1	A pointer to a string in SRAM
s2	A far pointer to a string in Flash
n	The maximum number of bytes to compare

Returns

The strncasecmp_PF() function returns an integer less than, equal to, or greater than zero if *s1* (or the first *n* bytes thereof) is found, respectively, to be less than, to match, or be greater than *s2*. The contents of RAMPZ SFR are undefined when the function returns.

21.20.3.86 strncat_P()

```
char * strncat_P (
    char * dest,
    const char * src,
    size_t len )
```

Concatenate two strings.

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

Returns

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

21.20.3.87 strncat_PF()

```
char * strncat_PF (
    char * dst,
    uint_farptr_t src,
    size_t n )
```

Concatenate two strings.

The strncat_PF() function is similar to strncat(), except that the *src* string must be located in program space (flash) and is addressed using a far pointer.

Parameters

dst	A pointer to the destination string in SRAM
src	A far pointer to the source string in Flash
n	The maximum number of bytes to append

Returns

The strncat_PF() function returns a pointer to the resulting string *dst*. The contents of RAMPZ SFR are undefined when the function returns.

21.20.3.88 strncmp_P()

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

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

Returns

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

21.20.3.89 strncmp_PF()

```
int strncmp_PF (
    const char * s1,
    uint_farptr_t s2,
    size_t n )
```

Compare two strings with limited length.

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

Parameters

s1	A pointer to the first string in SRAM
s2	A far pointer to the second string in Flash
n	The maximum number of bytes to compare

Returns

The strncmp_PF() function returns an integer less than, equal to, or greater than zero if *s1* (or the first *n* bytes thereof) is found, respectively, to be less than, to match, or be greater than *s2*. The contents of RAMPZ SFR are undefined when the function returns.

21.20.3.90 strncpy_P()

```
char * strncpy_P (
    char * dest,
    const char * src,
    size_t n )
```

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

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

Returns

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

21.20.3.91 strncpy_PF()

```
char * strncpy_PF (
    char * dst,
    uint_farptr_t src,
    size_t n )
```

Duplicate a string until a limited length.

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

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

Parameters

dst	A pointer to the destination string in SRAM
src	A far pointer to the source string in Flash
n	The maximum number of bytes to copy

Returns

The strncpy_PF() function returns a pointer to the destination string *dst*. The contents of RAMPZ SFR are undefined when the function returns.

21.20.3.92 strnlen_P()

```
size_t strnlen_P (
    const char * src,
    size_t len )
```

Determine the length of a fixed-size string.

The strnlen_P() function is similar to strnlen(), except that `src` is a pointer to a string in program space.

Returns

The strnlen_P function returns strlen_P(src), if that is less than `len`, or `len` if there is no '\0' character among the first `len` characters pointed to by `src`.

21.20.3.93 strnlen_PF()

```
size_t strnlen_PF (
    uint_farptr_t s,
    size_t len )
```

Determine the length of a fixed-size string.

The strnlen_PF() function is similar to strnlen(), except that *s* is a far pointer to a string in program space.

Parameters

s	A far pointer to the string in Flash
len	The maximum number of length to return

Returns

The strnlen_PF function returns strlen_P(*s*), if that is less than *len*, or *len* if there is no '\0' character among the first *len* characters pointed to by *s*. The contents of RAMPZ SFR are undefined when the function returns.

21.20.3.94 strpbrk_P()

```
char * strpbrk_P (
    const char * s,
    const char * accept )
```

The strpbrk_P() function locates the first occurrence in the string `s` of any of the characters in the flash string `accept`. This function is similar to strpbrk() except that `accept` is a pointer to a string in program space.

Returns

The strpbrk_P() function returns a pointer to the character in `s` that matches one of the characters in `accept`, or `NULL` if no such character is found. The terminating zero is not considered as a part of string: if one or both args are empty, the result will `NULL`.

21.20.3.95 strrchr_P()

```
const char * strrchr_P (
    const char * s,
    int val )
```

Locate character in string.

The strrchr_P() function returns a pointer to the last occurrence of the character `val` in the flash string `s`.

Returns

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

21.20.3.96 strsep_P()

```
char * strsep_P (
    char ** sp,
    const char * delim )
```

Parse a string into tokens.

The strsep_P() function locates, in the string referenced by `*sp`, the first occurrence of any character in the string `delim` (or the terminating '\0' character) and replaces it with a '\0'. The location of the next character after the delimiter character (or `NULL`, if the end of the string was reached) is stored in `*sp`. An ``empty'' field, i.e. one caused by two adjacent delimiter characters, can be detected by comparing the location referenced by the pointer returned in `*sp` to '\0'. This function is similar to strsep() except that `delim` is a pointer to a string in program space.

Returns

The strsep_P() function returns a pointer to the original value of `*sp`. If `*sp` is initially `NULL`, strsep_P() returns `NULL`.

21.20.3.97 strspn_P()

```
size_t strspn_P (
    const char * s,
    const char * accept )
```

The strspn_P() function calculates the length of the initial segment of `s` which consists entirely of characters in `accept`. This function is similar to strspn() except that `accept` is a pointer to a string in program space.

Returns

The strspn_P() function returns the number of characters in the initial segment of `s` which consist only of characters from `accept`. The terminating zero is not considered as a part of string.

21.20.3.98 strstr_P()

```
char * strstr_P (
    const char * s1,
    const char * s2 )
```

Locate a substring.

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

Returns

The strstr_P() function returns a pointer to the beginning of the substring, or NULL if the substring is not found. If `s2` points to a string of zero length, the function returns `s1`.

21.20.3.99 strstr_PF()

```
char * strstr_PF (
    const char * s1,
    uint_farptr_t s2 )
```

Locate a substring.

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

Returns

The strstr_PF() function returns a pointer to the beginning of the substring, or NULL if the substring is not found. If `s2` points to a string of zero length, the function returns `s1`. The contents of RAMPZ SFR are undefined when the function returns.

21.20.3.100 strtok_P()

```
char * strtok_P (
    char * s,
    const char * delim )
```

Parses the string into tokens.

strtok_P() parses the string `s` into tokens. The first call to strtok_P() should have `s` as its first argument. Subsequent calls should have the first argument set to NULL. If a token ends with a delimiter, this delimiting character is overwritten with a '\0' and a pointer to the next character is saved for the next call to strtok_P(). The delimiter string `delim` may be different for each call.

The strtok_P() function is similar to strtok() except that `delim` is pointer to a string in program space.

Returns

The strtok_P() function returns a pointer to the next token or NULL when no more tokens are found.

Note

strtok_P() is NOT reentrant. For a reentrant version of this function see strtok_rP().

21.20.3.101 strtok_rP()

```
char * strtok_rP (
    char * string,
    const char * delim,
    char ** last )
```

Parses string into tokens.

The strtok_rP() function parses `string` into tokens. The first call to strtok_rP() should have string as its first argument. Subsequent calls should have the first argument set to NULL. If a token ends with a delimiter, this delimiting character is overwritten with a '\0' and a pointer to the next character is saved for the next call to strtok_rP(). The delimiter string `delim` may be different for each call. `last` is a user allocated char* pointer. It must be the same while parsing the same string. strtok_rP() is a reentrant version of strtok_P().

The strtok_rP() function is similar to strtok_r() except that `delim` is pointer to a string in program space.

Returns

The strtok_rP() function returns a pointer to the next token or NULL when no more tokens are found.

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

Macros

- #define clock_prescale_get() (clock_div_t)(CLKPR & (uint8_t)((1<<CLKPS0)|(1<<CLKPS1)|(1<<CLKPS2)|(1<<CLKPS3))

Functions

- static void power_all_enable ()
- static void power_all_disable ()
- void clock_prescale_set (clock_div_t __x)

21.21.1 Detailed Description

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

Many AVRs contain a Power Reduction Register (PRR) or Registers (PRRx) that allow you to reduce power consumption by disabling or enabling various on-board peripherals as needed. Some devices have the XTAL Divide Control Register (XDIV) which offer similar functionality as System Clock Prescale Register (CLKPR).

There are many macros in this header file that provide an easy interface to enable or disable on-board peripherals to reduce power. See the table below.

Note

Not all AVR devices have a Power Reduction Register (for example the ATmega8). On those devices without a Power Reduction Register, the power reduction macros are not available..

Not all AVR devices contain the same peripherals (for example, the LCD interface), or they will be named differently (for example, USART and USART0). Please consult your device's datasheet, or the header file, to find out which macros are applicable to your device.

For device using the XTAL Divide Control Register (XDIV), when prescaler is used, Timer/Counter0 can only be used in asynchronous mode. Keep in mind that Timer/Counter0 source shall be less than ¼th of peripheral clock. Therefore, when using a typical 32.768 kHz crystal, one shall not scale the clock below 131.072 kHz.

Table 39 Power Macros

Power Macro	Description
power_aca_disable()	Disable the Analog Comparator on PortA
power_aca_enable()	Enable the Analog Comparator on PortA
power_adc_enable()	Enable the Analog to Digital Converter module
power_adc_disable()	Disable the Analog to Digital Converter module
power_adca_disable()	Disable the Analog to Digital Converter module on PortA
power_adca_enable()	Enable the Analog to Digital Converter module on PortA
power_evsys_disable()	Disable the EVSYS module
power_evsys_enable()	Enable the EVSYS module
power_hiresc_disable()	Disable the HIRES module on PortC
power_hiresc_enable()	Enable the HIRES module on PortC
power_lcd_enable()	Enable the LCD module
power_lcd_disable()	Disable the LCD module
power_pga_enable()	Enable the Programmable Gain Amplifier module
power_pga_disable()	Disable the Programmable Gain Amplifier module
power_pscr_enable()	Enable the Reduced Power Stage Controller module
power_pscr_disable()	Disable the Reduced Power Stage Controller module
power_psc0_enable()	Enable the Power Stage Controller 0 module
power_psc0_disable()	Disable the Power Stage Controller 0 module

Power Macro	Description
`power_psc1_enable()`	Enable the Power Stage Controller 1 module
`power_psc1_disable()`	Disable the Power Stage Controller 1 module
`power_psc2_enable()`	Enable the Power Stage Controller 2 module
`power_psc2_disable()`	Disable the Power Stage Controller 2 module
`power_ram0_enable()`	Enable the SRAM block 0
`power_ram0_disable()`	Disable the SRAM block 0
`power_ram1_enable()`	Enable the SRAM block 1
`power_ram1_disable()`	Disable the SRAM block 1
`power_ram2_enable()`	Enable the SRAM block 2
`power_ram2_disable()`	Disable the SRAM block 2
`power_ram3_enable()`	Enable the SRAM block 3
`power_ram3_disable()`	Disable the SRAM block 3
`power_rtc_disable()`	Disable the RTC module
`power_rtc_enable()`	Enable the RTC module
`power_spi_enable()`	Enable the Serial Peripheral Interface module
`power_spi_disable()`	Disable the Serial Peripheral Interface module
`power_spic_disable()`	Disable the SPI module on PortC
`power_spic_enable()`	Enable the SPI module on PortC
`power_spid_disable()`	Disable the SPI module on PortD
`power_spid_enable()`	Enable the SPI module on PortD
`power_tc0c_disable()`	Disable the TC0 module on PortC
`power_tc0c_enable()`	Enable the TC0 module on PortC
`power_tc0d_disable()`	Disable the TC0 module on PortD
`power_tc0d_enable()`	Enable the TC0 module on PortD
`power_tc0e_disable()`	Disable the TC0 module on PortE
`power_tc0e_enable()`	Enable the TC0 module on PortE
`power_tc0f_disable()`	Disable the TC0 module on PortF
`power_tc0f_enable()`	Enable the TC0 module on PortF
`power_tc1c_disable()`	Disable the TC1 module on PortC
`power_tc1c_enable()`	Enable the TC1 module on PortC
`power_twic_disable()`	Disable the Two Wire Interface module on PortC
`power_twic_enable()`	Enable the Two Wire Interface module on PortC
`power_twie_disable()`	Disable the Two Wire Interface module on PortE
`power_twie_enable()`	Enable the Two Wire Interface module on PortE
`power_timer0_enable()`	Enable the Timer 0 module
`power_timer0_disable()`	Disable the Timer 0 module
`power_timer1_enable()`	Enable the Timer 1 module
`power_timer1_disable()`	Disable the Timer 1 module
`power_timer2_enable()`	Enable the Timer 2 module
`power_timer2_disable()`	Disable the Timer 2 module
`power_timer3_enable()`	Enable the Timer 3 module
`power_timer3_disable()`	Disable the Timer 3 module
`power_timer4_enable()`	Enable the Timer 4 module
`power_timer4_disable()`	Disable the Timer 4 module
`power_timer5_enable()`	Enable the Timer 5 module
`power_timer5_disable()`	Disable the Timer 5 module
`power_twi_enable()`	Enable the Two Wire Interface module
`power_twi_disable()`	Disable the Two Wire Interface module
`power_usart_enable()`	Enable the USART module
`power_usart_disable()`	Disable the USART module
`power_usart0_enable()`	Enable the USART 0 module
`power_usart0_disable()`	Disable the USART 0 module
`power_usart1_enable()`	Enable the USART 1 module
`power_usart1_disable()`	Disable the USART 1 module
`power_usart2_enable()`	Enable the USART 2 module
`power_usart2_disable()`	Disable the USART 2 module

Power Macro	Description
power_usart3_enable()	Enable the USART 3 module
power_usart3_disable()	Disable the USART 3 module
power_usartc0_disable()	Disable the USART0 module on PortC
power_usartc0_enable()	Enable the USART0 module on PortC
power_usartd0_disable()	Disable the USART0 module on PortD
power_usartd0_enable()	Enable the USART0 module on PortD
power_usarte0_disable()	Disable the USART0 module on PortE
power_usarte0_enable()	Enable the USART0 module on PortE
power_usartf0_disable()	Disable the USART0 module on PortF
power_usartf0_enable()	Enable the USART0 module on PortF
power_usb_enable()	Enable the USB module
power_usb_disable()	Disable the USB module
power_usi_enable()	Enable the Universal Serial Interface module
power_usi_disable()	Disable the Universal Serial Interface module
power_vadc_enable()	Enable the Voltage ADC module
power_vadc_disable()	Disable the Voltage ADC module
power_all_enable()	Enable all modules
power_all_disable()	Disable all modules

Some of the newer AVRs contain a System Clock Prescale Register (CLKPR) that allows you to decrease the system clock frequency and the power consumption when the need for processing power is low. On some earlier AVRs (ATmega103, ATmega64, ATmega128), similar functionality can be achieved through the XTAL Divide Control Register. Below are two macros and an enumerated type that can be used to interface to the Clock Prescale Register or XTAL Divide Control Register.

Note

Not all AVR devices have a clock prescaler. On those devices without a Clock Prescale Register or XTAL Divide Control Register, these macros are not available.

```
typedef enum
{
    clock_div_1 = 0,
    clock_div_2 = 1,
    clock_div_4 = 2,
    clock_div_8 = 3,
    clock_div_16 = 4,
    clock_div_32 = 5,
    clock_div_64 = 6,
    clock_div_128 = 7,
    clock_div_256 = 8,
    clock_div_1_rc = 15, // ATmega128RFA1 only
} clock_div_t;
```

Clock prescaler setting enumerations for device using System Clock Prescale Register.

```
typedef enum
{
    clock_div_1 = 1,
    clock_div_2 = 2,
    clock_div_4 = 4,
    clock_div_8 = 8,
    clock_div_16 = 16,
    clock_div_32 = 32,
    clock_div_64 = 64,
    clock_div_128 = 128
} clock_div_t;
```

Clock prescaler setting enumerations for device using XTAL Divide Control Register.

21.21.2 Macro Definition Documentation

21.21.2.1 clock_prescale_get `#define clock_prescale_get( ) (clock_div_t)(CLKPR & (uint8_t)((1<<CLKPS0)|(1<<CL`

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

Note

For device with XTAL Divide Control Register (XDIV), return can actually range from 1 to 129. Care should be taken has the return value could differ from the typedef enum clock_div_t. This should only happen if clock_prescale_set was previously called with a value other than those defined by `clock_div_t`.

21.21.3 Function Documentation

21.21.3.1 clock_prescale_set() `clock_prescale_set ( clock_div_t x )`

Set the clock prescaler register select bits, selecting a system clock division setting. This function is inlined, even if compiler optimizations are disabled.

The type of `x` is `clock_div_t`.

Note

For device with XTAL Divide Control Register (XDIV), `x` can actually range from 1 to 129. Thus, one does not need to use `clock_div_t` type as argument.

21.21.3.2 power_all_disable() `void power_all_disable ( ) [inline], [static]`

Disable all modules.

21.21.3.3 power_all_enable() `void power_all_enable ( ) [inline], [static]`

Enable all modules.

21.22 Additional notes from <avr/sfr_defs.h>

The <avr/sfr_defs.h> file is included by all of the <avr/ioXXXX.h> files, which use macros defined here to make the special function register definitions look like C variables or simple constants, depending on the _SFR_ASM_COMPAT define. Some examples from <avr/iocanxx.h> to show how to define such macros:

```
#define PORTA   _SFR_IO8(0x02)
#define EEAR    _SFR_IO16(0x21)
#define UDR0    _SFR_MEM8(0xC6)
#define TCNT3   _SFR_MEM16(0x94)
#define CANIDT  _SFR_MEM32(0xF0)
```

If _SFR_ASM_COMPAT is not defined, C programs can use names like PORTA directly in C expressions (also on the left side of assignment operators) and GCC will do the right thing (use short I/O instructions if possible). The __SFR_OFFSET definition is not used in any way in this case.

Define _SFR_ASM_COMPAT as 1 to make these names work as simple constants (addresses of the I/O registers). This is necessary when included in preprocessed assembler (*.S) source files, so it is done automatically if __ASSEMBLER__ is defined. By default, all addresses are defined as if they were memory addresses (used in lds/sts instructions). To use these addresses in in/out instructions, you must subtract 0x20 from them.

For more backwards compatibility, insert the following at the start of your old assembler source file:

```
#define __SFR_OFFSET 0
```

This automatically subtracts 0x20 from I/O space addresses, but it's a hack, so it is recommended to change your source: wrap such addresses in macros defined here, as shown below. After this is done, the __SFR_OFFSET definition is no longer necessary and can be removed.

Real example - this code could be used in a boot loader that is portable between devices with SPMCR at different addresses.

```
<avr/iom163.h>: #define SPMCR _SFR_IO8(0x37)
<avr/iom128.h>: #define SPMCR _SFR_MEM8(0x68)

#if _SFR_IO_REG_P(SPMCR)
    out _SFR_IO_ADDR(SPMCR), r24
#else
    sts _SFR_MEM_ADDR(SPMCR), r24
#endif
```

You can use the in/out/cbi/sbi/sbic/sbis instructions, without the _SFR_IO_REG_P test, if you know that the register is in the I/O space (as with SREG, for example). If it isn't, the assembler will complain (I/O address out of range 0...0x3f), so this should be fairly safe.

If you do not define __SFR_OFFSET (so it will be 0x20 by default), all special register addresses are defined as memory addresses (so SREG is 0x5f), and (if code size and speed are not important, and you don't like the ugly #if above) you can always use lds/sts to access them. But, this will not work if __SFR_OFFSET != 0x20, so use a different macro (defined only if __SFR_OFFSET == 0x20) for safety:

```
sts _SFR_ADDR(SPMCR), r24
```

In C programs, all 3 combinations of _SFR_ASM_COMPAT and __SFR_OFFSET are supported - the _SFR_ADDR(SPMCR) macro can be used to get the address of the SPMCR register (0x57 or 0x68 depending on device).

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

Modules

- Additional notes from <avr/sfr_defs.h>

Bit manipulation

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

IO register bit manipulation

- #define bit_is_set(sfr, bit) (_SFR_BYTE(sfr) & _BV(bit))
- #define bit_is_clear(sfr, bit) (!(_SFR_BYTE(sfr) & _BV(bit)))
- #define loop_until_bit_is_set(sfr, bit) do { } while (bit_is_clear(sfr, bit))
- #define loop_until_bit_is_clear(sfr, bit) do { } while (bit_is_set(sfr, bit))

21.23.1 Detailed Description

When working with microcontrollers, many tasks usually consist of controlling internal peripherals, or external peripherals that are connected to the device. The entire IO address space is made available as *memory-mapped IO*, i.e. it can be accessed using all the MCU instructions that are applicable to normal data memory. For most AVR devices, the IO register space is mapped into the data memory address space with an offset of 0x20 since the bottom of this space is reserved for direct access to the MCU registers. (Actual SRAM is available only behind the IO register area, starting at some specific address depending on the device.)

For example the user can access memory-mapped IO registers as if they were globally defined variables like this:

```
PORTA = 0x33;
unsigned char foo = PINA;
```

The compiler will choose the correct instruction sequence to generate based on the address of the register being accessed.

The advantage of using the memory-mapped registers in C programs is that it makes the programs more portable to other C compilers for the AVR platform.

Note that special care must be taken when accessing some of the 16-bit timer IO registers where access from both the main program and within an interrupt context can happen. See Why do some 16-bit timer registers sometimes get trashed?.

Porting programs that use the deprecated sbi/cbi macros

Access to the AVR single bit set and clear instructions are provided via the standard C bit manipulation commands. The sbi and cbi macros are no longer directly supported. sbi (sfr,bit) can be replaced by sfr |= _BV(bit) .

i.e.: sbi(PORTB, PB1); is now PORTB |= _BV(PB1);

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

Likewise, cbi (sfr,bit) is now sfr &= ~(_BV(bit));

21.23.2 Macro Definition Documentation

21.23.2.1 _BV

```
#define _BV(
    bit ) (1 << (bit))
#include <avr/io.h>
```

Converts a bit number into a byte value.

Note

The bit shift is performed by the compiler which then inserts the result into the code. Thus, there is no run-time overhead when using _BV().

21.23.2.2 bit_is_clear

```
#define bit_is_clear(
    sfr,
    bit ) (!(_SFR_BYTE(sfr) & _BV(bit)))
#include <avr/io.h>
```

Test whether bit `bit` in IO register `sfr` is clear. This will return non-zero if the bit is clear, and a 0 if the bit is set.

21.23.2.3 bit_is_set

```
#define bit_is_set(
    sfr,
    bit ) (_SFR_BYTE(sfr) & _BV(bit))
#include <avr/io.h>
```

Test whether bit `bit` in IO register `sfr` is set. This will return a 0 if the bit is clear, and non-zero if the bit is set.

21.23.2.4 loop_until_bit_is_clear

```
#define loop_until_bit_is_clear(
    sfr,
    bit ) do { } while (bit_is_set(sfr, bit))
#include <avr/io.h>
```

Wait until bit `bit` in IO register `sfr` is clear.

21.23.2.5 loop_until_bit_is_set

```
#define loop_until_bit_is_set(
    sfr,
    bit ) do { } while (bit_is_clear(sfr, bit))
#include <avr/io.h>
```

Wait until bit `bit` in IO register `sfr` is set.

21.24 <avr/signature.h>: Signature Support

Introduction

The <avr/signature.h> header file allows the user to automatically and easily include the device's signature data in a special section of the final linked ELF file.

This value can then be used by programming software to compare the on-device signature with the signature recorded in the ELF file to look for a match before programming the device.

API Usage Example

Usage is very simple; just include the header file:

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

This will declare a constant unsigned char array and it is initialized with the three signature bytes, MSB first, that are defined in the device I/O header file. This array is then placed in the .signature section in the resulting linked ELF file.

The three signature bytes that are used to initialize the array are these defined macros in the device I/O header file, from MSB to LSB: SIGNATURE_2, SIGNATURE_1, SIGNATURE_0.

This header file should only be included once in an application.

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

Functions

- void sleep_enable (void)
- void sleep_disable (void)
- void sleep_cpu (void)
- void sleep_mode (void)
- void sleep_bod_disable (void)

21.25.1 Detailed Description

21.25.2 Function Documentation

21.25.2.1 sleep_bod_disable()

```
void sleep_bod_disable (
        void )
```

Disable BOD before going to sleep. Not available on all devices.

21.25.2.2 sleep_cpu()

```
void sleep_cpu (
        void )
```

Put the device into sleep mode. The SE bit must be set beforehand, and it is recommended to clear it afterwards.

21.25.2.3 sleep_disable()

```
void sleep_disable (
        void )
```

Clear the SE (sleep enable) bit.

21.25.2.4 sleep_enable() void sleep_enable (void)

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

Use of the SLEEP instruction can allow an application to reduce its power comsumption considerably. AVR devices can be put into different sleep modes. Refer to the datasheet for the details relating to the device you are using.

There are several macros provided in this header file to actually put the device into sleep mode. The simplest way is to optionally set the desired sleep mode using set_sleep_mode() (it usually defaults to idle mode where the CPU is put on sleep but all peripheral clocks are still running), and then call sleep_mode(). This macro automatically sets the sleep enable bit, goes to sleep, and clears the sleep enable bit.

Example:

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

...
  set_sleep_mode(<mode>);
  sleep_mode();
```

Note that unless your purpose is to completely lock the CPU (until a hardware reset), interrupts need to be enabled before going to sleep.

As the sleep_mode() macro might cause race conditions in some situations, the individual steps of manipulating the sleep enable (SE) bit, and actually issuing the SLEEP instruction, are provided in the macros sleep_enable(), sleep_disable(), and sleep_cpu(). This also allows for test-and-sleep scenarios that take care of not missing the interrupt that will awake the device from sleep.

Example:

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

...
  set_sleep_mode(<mode>);
  cli();
  if (some_condition)
  {
    sleep_enable();
    sei();
    sleep_cpu();
    sleep_disable();
  }
  sei();
```

This sequence ensures an atomic test of some_condition with interrupts being disabled. If the condition is met, sleep mode will be prepared, and the SLEEP instruction will be scheduled immediately after an SEI instruction. As the intruction right after the SEI is guaranteed to be executed before an interrupt could trigger, it is sure the device will really be put to sleep.

Some devices have the ability to disable the Brown Out Detector (BOD) before going to sleep. This will also reduce power while sleeping. If the specific AVR device has this ability then an additional macro is defined↵ : sleep_bod_disable(). This macro generates inlined assembly code that will correctly implement the timed sequence for disabling the BOD before sleeping. However, there is a limited number of cycles after the BOD has been disabled that the device can be put into sleep mode, otherwise the BOD will not truly be disabled. Recommended practice is to disable the BOD (sleep_bod_disable()), set the interrupts (sei()), and then put the device to sleep (sleep_cpu()), like so:

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

...
  set_sleep_mode(<mode>);
  cli();
  if (some_condition)
  {
    sleep_enable();
    sleep_bod_disable();
    sei();
    sleep_cpu();
    sleep_disable();
  }
  sei();
```

Put the device in sleep mode. How the device is brought out of sleep mode depends on the specific mode selected with the set_sleep_mode() function. See the data sheet for your device for more details.

Set the SE (sleep enable) bit.

21.25.2.5 sleep_mode()

```
void sleep_mode (
    void )
```

Put the device into sleep mode, taking care of setting the SE bit before, and clearing it afterwards.

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

Macros

- #define __AVR_LIBC_VERSION_STRING__ "2.2.0"
- #define __AVR_LIBC_VERSION__ 20200UL
- #define __AVR_LIBC_DATE_STRING__ "20240608"
- #define __AVR_LIBC_DATE_ 20240608UL
- #define __AVR_LIBC_MAJOR__ 2
- #define __AVR_LIBC_MINOR__ 2
- #define __AVR_LIBC_REVISION__ 0

21.26.1 Detailed Description

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

This header file defines macros that contain version numbers and strings describing the current version of avr-libc.

The version number itself basically consists of three pieces that are separated by a dot: the major number, the minor number, and the revision number. For development versions (which use an odd minor number), the string representation additionally gets the date code (YYYYMMDD) appended.

This file will also be included by <avr/io.h>. That way, portable tests can be implemented using <avr/io.h> that can be used in code that wants to remain backwards-compatible to library versions prior to the date when the library version API had been added, as referenced but undefined C preprocessor macros automatically evaluate to 0.

21.26.2 Macro Definition Documentation

21.26.2.1 __AVR_LIBC_DATE_

```
#define __AVR_LIBC_DATE_ 20240608UL
```

Numerical representation of the release date.

21.26.2.2 __AVR_LIBC_DATE_STRING__

```
#define __AVR_LIBC_DATE_STRING__ "20240608"
```

String literal representation of the release date.

21.26.2.3 __AVR_LIBC_MAJOR__

```
#define __AVR_LIBC_MAJOR__ 2
```

Library major version number.

21.26.2.4 __AVR_LIBC_MINOR__ `#define __AVR_LIBC_MINOR__ 2`

Library minor version number.

21.26.2.5 __AVR_LIBC_REVISION__ `#define __AVR_LIBC_REVISION__ 0`

Library revision number.

21.26.2.6 __AVR_LIBC_VERSION__ `#define __AVR_LIBC_VERSION__ 20200UL`

Numerical representation of the current library version.

In the numerical representation, the major number is multiplied by 10000, the minor number by 100, and all three parts are then added. It is intented to provide a monotonically increasing numerical value that can easily be used in numerical checks.

21.26.2.7 __AVR_LIBC_VERSION_STRING__ `#define __AVR_LIBC_VERSION_STRING__ "2.2.0"`

String literal representation of the current library version.

21.27 <avr/builtins.h>: avr-gcc builtins documentation

Functions

- void __builtin_avr_sei (void)
- void __builtin_avr_cli (void)
- void __builtin_avr_sleep (void)
- void __builtin_avr_wdr (void)
- uint8_t __builtin_avr_swap (uint8_t __b)
- uint16_t __builtin_avr_fmul (uint8_t __a, uint8_t __b)
- int16_t __builtin_avr_fmuls (int8_t __a, int8_t __b)
- int16_t __builtin_avr_fmulsu (int8_t __a, uint8_t __b)

21.27.1 Detailed Description

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

Note

> This file only documents some avr-gcc builtins. For functions built-in in the compiler, there should be no prototype declarations.

See also the GCC documentation for a full list of avr-gcc builtins.

21.27.2 Function Documentation

21.27.2.1 __builtin_avr_cli() void __builtin_avr_cli (void)

Disables all interrupts by clearing the global interrupt mask.

21.27.2.2 __builtin_avr_fmul() uint16_t __builtin_avr_fmul (uint8_t *__a*, uint8_t *__b*)

Emits an FMUL (fractional multiply unsigned) instruction.

21.27.2.3 __builtin_avr_fmuls() int16_t __builtin_avr_fmuls (int8_t *__a*, int8_t *__b*)

Emits an FMUL (fractional multiply signed) instruction.

21.27.2.4 __builtin_avr_fmulsu() int16_t __builtin_avr_fmulsu (int8_t *__a*, uint8_t *__b*)

Emits an FMUL (fractional multiply signed with unsigned) instruction.

21.27.2.5 __builtin_avr_sei() void __builtin_avr_sei (void)

Enables interrupts by setting the global interrupt mask.

21.27.2.6 __builtin_avr_sleep() void __builtin_avr_sleep (void)

Emits a `SLEEP` instruction.

21.27.2.7 __builtin_avr_swap() uint8_t __builtin_avr_swap (uint8_t *__b*)

Emits a SWAP (nibble swap) instruction on __b.

21.27.2.8 __builtin_avr_wdr() void __builtin_avr_wdr (void)

Emits a WDR (watchdog reset) instruction.

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

Macros

- #define wdt_reset() __asm__ __volatile__ ("wdr")
- #define wdt_enable(timeout)
- #define WDTO_15MS 0
- #define WDTO_30MS 1
- #define WDTO_60MS 2
- #define WDTO_120MS 3
- #define WDTO_250MS 4
- #define WDTO_500MS 5
- #define WDTO_1S 6
- #define WDTO_2S 7
- #define WDTO_4S 8
- #define WDTO_8S 9

21.28.1 Detailed Description

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

This header file declares the interface to some inline macros handling the watchdog timer present in many AVR devices. In order to prevent the watchdog timer configuration from being accidentally altered by a crashing application, a special timed sequence is required in order to change it. The macros within this header file handle the required sequence automatically before changing any value. Interrupts will be disabled during the manipulation.

Note

Depending on the fuse configuration of the particular device, further restrictions might apply, in particular it might be disallowed to turn off the watchdog timer.

Note that for newer devices (ATmega88 and newer, effectively any AVR that has the option to also generate interrupts), the watchdog timer remains active even after a system reset (except a power-on condition), using the fastest prescaler value (approximately 15 ms). It is therefore required to turn off the watchdog early during program startup, the datasheet recommends a sequence like the following:

```
#include <stdint.h>
#include <avr/wdt.h>

uint8_t mcusr_mirror __attribute__ ((section (".noinit")));

__attribute__((used, unused, naked, section(".init3")))
static void get_mcusr (void);

void get_mcusr (void)
{
  mcusr_mirror = MCUSR;
  MCUSR = 0;
  wdt_disable();
}
```

Saving the value of MCUSR in `mcusr_mirror` is only needed if the application later wants to examine the reset source, but in particular, clearing the watchdog reset flag before disabling the watchdog is required, according to the datasheet.

21.28.2 Macro Definition Documentation

21.28.2.1 wdt_enable `#define wdt_enable( timeout )`

Enable the watchdog timer, configuring it for expiry after `timeout` (which is a combination of the `WDP0` through `WDP2` bits to write into the `WDTCR` register; For those devices that have a `WDTCSR` register, it uses the combination of the `WDP0` through `WDP3` bits).

See also the symbolic constants `WDTO_15MS` et al.

21.28.2.2 wdt_reset `#define wdt_reset( ) __asm__ __volatile__ ("wdr")`

Reset the watchdog timer. When the watchdog timer is enabled, a call to this instruction is required before the timer expires, otherwise a watchdog-initiated device reset will occur.

21.28.2.3 WDTO_120MS `#define WDTO_120MS 3`

See `WDTO_15MS`

21.28.2.4 WDTO_15MS `#define WDTO_15MS 0`

Symbolic constants for the watchdog timeout. Since the watchdog timer is based on a free-running RC oscillator, the times are approximate only and apply to a supply voltage of 5 V. At lower supply voltages, the times will increase. For older devices, the times will be as large as three times when operating at Vcc = 3 V, while the newer devices (e. g. ATmega128, ATmega8) only experience a negligible change.

Possible timeout values are: 15 ms, 30 ms, 60 ms, 120 ms, 250 ms, 500 ms, 1 s, 2 s. (Some devices also allow for 4 s and 8 s.) Symbolic constants are formed by the prefix `WDTO_`, followed by the time.

Example that would select a watchdog timer expiry of approximately 500 ms:

```
wdt_enable(WDTO_500MS);
```

21.28.2.5 WDTO_1S `#define WDTO_1S 6`

See `WDTO_15MS`

21.28.2.6 WDTO_250MS `#define WDTO_250MS 4`

See `WDTO_15MS`

21.28.2.7 WDTO_2S `#define WDTO_2S 7`

See `WDTO_15MS`

21.28.2.8 WDTO_30MS `#define WDTO_30MS 1`

See `WDTO_15MS`

21.28.2.9 WDTO_4S `#define WDTO_4S 8`

See `WDTO_15MS` Note: This is only available on the ATtiny2313, ATtiny24, ATtiny44, ATtiny84, ATtiny84A, ATtiny25, ATtiny45, ATtiny85, ATtiny261, ATtiny461, ATtiny861, ATmega48*, ATmega88*, ATmega168*, ATmega328*, ATmega164P, ATmega324P, ATmega324PB, ATmega644P, ATmega644, ATmega640, ATmega1280, ATmega1281, ATmega2560, ATmega2561, ATmega8HVA, ATmega16HVA, ATmega32HVB, ATmega406, ATmega1284P, AT90PWM1, AT90PWM2, AT90PWM2B, AT90PWM3, AT90PWM3B, AT90PWM216, AT90PWM316, AT90PWM81, AT90PWM161, AT90USB82, AT90USB162, AT90USB646, AT90USB647, AT90USB1286, AT90USB1287, ATtiny48, ATtiny88.

Note: This value does *not* match the bit pattern of the respective control register. It is solely meant to be used together with wdt_enable().

21.28.2.10 WDTO_500MS `#define WDTO_500MS 5`

See `WDTO_15MS`

21.28.2.11 WDTO_60MS `#define WDTO_60MS 2`

See `WDTO_15MS`

21.28.2.12 WDTO_8S `#define WDTO_8S 9`

See `WDTO_15MS` Note: This is only available on the ATtiny2313, ATtiny24, ATtiny44, ATtiny84, ATtiny84A, ATtiny25, ATtiny45, ATtiny85, ATtiny261, ATtiny461, ATtiny861, ATmega48*, ATmega88*, ATmega168*, ATmega328*, ATmega164P, ATmega324P, ATmega324PB, ATmega644P, ATmega644, ATmega640, ATmega1280, ATmega1281, ATmega2560, ATmega2561, ATmega8HVA, ATmega16HVA, ATmega32HVB, ATmega406, ATmega1284P, ATmega2564RFR2, ATmega256RFR2, ATmega1284RFR2, ATmega128RFR2, ATmega644RFR2, ATmega64RFR2 AT90PWM1, AT90PWM2, AT90PWM2B, AT90PWM3, AT90PWM3B, AT90PWM216, AT90PWM316, AT90PWM81, AT90PWM161, AT90USB82, AT90USB162, AT90USB646, AT90USB647, AT90USB1286, AT90USB1287, ATtiny48, ATtiny88, ATxmega16a4u, ATxmega32a4u, ATxmega16c4, ATxmega32c4, ATxmega128c3, ATxmega192c3, ATxmega256c3.

Note: This value does *not* match the bit pattern of the respective control register. It is solely meant to be used together with wdt_enable().

21.29 <util/delay.h>: Convenience functions for busy-wait delay loops

Macros

- #define F_CPU 1000000UL

Functions

- static void _delay_ms (double __ms)
- static void _delay_us (double __us)

21.29.1 Detailed Description

```
#define F_CPU 1000000UL  // 1 MHz
//#define F_CPU 14.7456e6
#include <util/delay.h>
```

Note

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

The functions in this header file are wrappers around the basic busy-wait functions from <`util/delay_basic.h`>. They are meant as convenience functions where actual time values can be specified rather than a number of cycles to wait for. The idea behind is that compile-time constant expressions will be eliminated by compiler optimization so floating-point expressions can be used to calculate the number of delay cycles needed based on the CPU frequency passed by the macro F_CPU.

Note

In order for these functions to work as intended, compiler optimizations *must* be enabled, and the delay time *must* be an expression that is a known constant at compile-time. If these requirements are not met, the resulting delay will be much longer (and basically unpredictable), and applications that otherwise do not use floating-point calculations will experience severe code bloat by the floating-point library routines linked into the application.

The functions available allow the specification of microsecond, and millisecond delays directly, using the application-supplied macro F_CPU as the CPU clock frequency (in Hertz).

21.29.2 Macro Definition Documentation

21.29.2.1 F_CPU `#define F_CPU 1000000UL`

CPU frequency in Hz.

The macro F_CPU specifies the CPU frequency to be considered by the delay macros. This macro is normally supplied by the environment (e.g. from within a project header, or the project's Makefile). The value 1 MHz here is only provided as a "vanilla" fallback if no such user-provided definition could be found.

In terms of the delay functions, the CPU frequency can be given as a floating-point constant (e.g. 3.6864e6 for 3.6864 MHz). However, the macros in <util/setbaud.h> require it to be an integer value.

21.29.3 Function Documentation

21.29.3.1 _delay_ms()

```
void _delay_ms (
    double __ms ) [inline], [static]
```

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

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

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

When the user request delay which exceed the maximum possible one, _delay_ms() provides a decreased resolution functionality. In this mode _delay_ms() will work with a resolution of 1/10 ms, providing delays up to 6.5535 seconds (independent from CPU frequency). The user will not be informed about decreased resolution.

If the avr-gcc toolchain has `__builtin_avr_delay_cycles()` support, maximal possible delay is 4294967.295 ms/ F_CPU in MHz. For values greater than the maximal possible delay, overflow may result in no delay i.e., 0 ms.

Conversion of `__ms` into clock cycles may not always result in an integral value. By default, the clock cycles are rounded up to the next integer. This ensures that the user gets at least `__ms` microseconds of delay.

Alternatively, by defining the macro `__DELAY_ROUND_DOWN__`, or `__DELAY_ROUND_CLOSEST__`, before including this header file, the algorithm can be made to round down, or round to closest integer, respectively.

Note

> The implementation of _delay_ms() based on `__builtin_avr_delay_cycles()` is not backward compatible with older implementations. In order to get functionality backward compatible with previous versions, the macro `__DELAY_BACKWARD_COMPATIBLE__` must be defined before including this header file.

21.29.3.2 _delay_us()

```
void _delay_us (
    double __us ) [inline], [static]
```

Perform a delay of `__us` microseconds, using _delay_loop_1().

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

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

If the user requests a delay greater than the maximal possible one, _delay_us() will automatically call _delay_ms() instead. The user will not be informed about this case.

If the avr-gcc toolchain has __builtin_avr_delay_cycles() support, maximal possible delay is 4294967.295 μs/ F_↩CPU in MHz. For values greater than the maximal possible delay, overflow may result in no delay i.e., 0 μs.

Conversion of `__us` into clock cycles may not always result in integer. By default, the clock cycles are rounded up to next integer. This ensures that the user gets at least `__us` microseconds of delay.

Alternatively, by defining the macro `__DELAY_ROUND_DOWN__`, or `__DELAY_ROUND_CLOSEST__`, before including this header file, the algorithm can be made to round down, or round to closest integer, respectively.

Note

> The implementation of _delay_us() based on `__builtin_avr_delay_cycles()` is not backward compatible with older implementations. In order to get functionality backward compatible with previous versions, the macro `__DELAY_BACKWARD_COMPATIBLE__` must be defined before including this header file.

21.30 <util/atomic.h> Atomically and Non-Atomically Executed Code Blocks

Macros

- #define ATOMIC_BLOCK(type)
- #define NONATOMIC_BLOCK(type)
- #define ATOMIC_RESTORESTATE
- #define ATOMIC_FORCEON
- #define NONATOMIC_RESTORESTATE
- #define NONATOMIC_FORCEOFF

21.30.1 Detailed Description

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

Note

The macros in this header file require the ISO/IEC 9899:1999 ("ISO C99") feature of for loop variables that are declared inside the for loop itself. For that reason, this header file can only be used if the standard level of the compiler (option –std=) is set to either `c99`, `gnu99` or higher.

The macros in this header file deal with code blocks that are guaranteed to be executed Atomically or Non-↩Atomically. The term "Atomic" in this context refers to the inability of the respective code to be interrupted.

These macros operate via automatic manipulation of the Global Interrupt Status (I) bit of the SREG register. Exit paths from both block types are all managed automatically without the need for special considerations, i.e. the interrupt status will be restored to the same value it had when entering the respective block (unless ATOMIC_↩FORCEON or NONATOMIC_FORCEOFF are used).

Warning

The features in this header are implemented by means of a for loop. This means that commands like `break` and `continue` that are located in an atomic block refer to the atomic for loop, not to a loop construct that hosts the atomic block.

A typical example that requires atomic access is a 16 (or more) bit variable that is shared between the main execution path and an ISR. While declaring such a variable as volatile ensures that the compiler will not optimize accesses to it away, it does not guarantee atomic access to it. Assuming the following example:

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

volatile uint16_t ctr;

ISR(TIMER1_OVF_vect)
{
  ctr--;
}

...
int
main(void)
{
   ...
   ctr = 0x200;
   start_timer();
   while (ctr != 0)
     // wait
       ;
   ...
}
```

There is a chance where the main context will exit its wait loop when the variable `ctr` just reached the value 0xFF. This happens because the compiler cannot natively access a 16-bit variable atomically in an 8-bit CPU. So the variable is for example at 0x100, the compiler then tests the low byte for 0, which succeeds. It then proceeds to test the high byte, but that moment the ISR triggers, and the main context is interrupted. The ISR will decrement the variable from 0x100 to 0xFF, and the main context proceeds. It now tests the high byte of the variable which is (now) also 0, so it concludes the variable has reached 0, and terminates the loop.

Using the macros from this header file, the above code can be rewritten like:

```
#include <stdint.h>
#include <avr/interrupt.h>
#include <avr/io.h>
#include <util/atomic.h>

volatile uint16_t ctr;

ISR(TIMER1_OVF_vect)
{
  ctr--;
}

...
int
main(void)
{
   ...
   ctr = 0x200;
   start_timer();
   sei();
   uint16_t ctr_copy;
   do
   {
     ATOMIC_BLOCK(ATOMIC_FORCEON)
     {
       ctr_copy = ctr;
     }
   }
   while (ctr_copy != 0);
   ...
}
```

This will install the appropriate interrupt protection before accessing variable `ctr`, so it is guaranteed to be consistently tested. If the global interrupt state were uncertain before entering the ATOMIC_BLOCK, it should be executed with the parameter ATOMIC_RESTORESTATE rather than ATOMIC_FORCEON.

See Problems with reordering code for things to be taken into account with respect to compiler optimizations.

21.30.2 Macro Definition Documentation

21.30.2.1 ATOMIC_BLOCK `#define ATOMIC_BLOCK( type )`

Creates a block of code that is guaranteed to be executed atomically. Upon entering the block the Global Interrupt Status flag in SREG is disabled, and re-enabled upon exiting the block from any exit path.

Two possible macro parameters are permitted, ATOMIC_RESTORESTATE and ATOMIC_FORCEON.

21.30.2.2 ATOMIC_FORCEON `#define ATOMIC_FORCEON`

This is a possible parameter for ATOMIC_BLOCK. When used, it will cause the ATOMIC_BLOCK to force the state of the SREG register on exit, enabling the Global Interrupt Status flag bit. This saves a small amount of flash space, a register, and one or more processor cycles, since the previous value of the SREG register does not need to be saved at the start of the block.

Care should be taken that ATOMIC_FORCEON is only used when it is known that interrupts are enabled before the block's execution or when the side effects of enabling global interrupts at the block's completion are known and understood.

21.30.2.3 ATOMIC_RESTORESTATE `#define ATOMIC_RESTORESTATE`

This is a possible parameter for ATOMIC_BLOCK. When used, it will cause the ATOMIC_BLOCK to restore the previous state of the SREG register, saved before the Global Interrupt Status flag bit was disabled. The net effect of this is to make the ATOMIC_BLOCK's contents guaranteed atomic, without changing the state of the Global Interrupt Status flag when execution of the block completes.

21.30.2.4 NONATOMIC_BLOCK `#define NONATOMIC_BLOCK( type )`

Creates a block of code that is executed non-atomically. Upon entering the block the Global Interrupt Status flag in SREG is enabled, and disabled upon exiting the block from any exit path. This is useful when nested inside ATOMIC_BLOCK sections, allowing for non-atomic execution of small blocks of code while maintaining the atomic access of the other sections of the parent ATOMIC_BLOCK.

Two possible macro parameters are permitted, NONATOMIC_RESTORESTATE and NONATOMIC_FORCEOFF.

21.30.2.5 NONATOMIC_FORCEOFF `#define NONATOMIC_FORCEOFF`

This is a possible parameter for NONATOMIC_BLOCK. When used, it will cause the NONATOMIC_BLOCK to force the state of the SREG register on exit, disabling the Global Interrupt Status flag bit. This saves a small amout of flash space, a register, and one or more processor cycles, since the previous value of the SREG register does not need to be saved at the start of the block.

Care should be taken that NONATOMIC_FORCEOFF is only used when it is known that interrupts are disabled before the block's execution or when the side effects of disabling global interrupts at the block's completion are known and understood.

21.30.2.6 NONATOMIC_RESTORESTATE `#define NONATOMIC_RESTORESTATE`

This is a possible parameter for NONATOMIC_BLOCK. When used, it will cause the NONATOMIC_BLOCK to restore the previous state of the SREG register, saved before the Global Interrupt Status flag bit was enabled. The net effect of this is to make the NONATOMIC_BLOCK's contents guaranteed non-atomic, without changing the state of the Global Interrupt Status flag when execution of the block completes.

21.31 <util/crc16.h>: CRC Computations

Functions

- static uint16_t _crc16_update (uint16_t __crc, uint8_t __data)
- static uint16_t _crc_xmodem_update (uint16_t __crc, uint8_t __data)
- static uint16_t _crc_ccitt_update (uint16_t __crc, uint8_t __data)
- static uint8_t _crc_ibutton_update (uint8_t __crc, uint8_t __data)
- static uint8_t _crc8_ccitt_update (uint8_t __crc, uint8_t __data)

21.31.1 Detailed Description

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

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

References:

See the Dallas Semiconductor app note 27 for 8051 assembler example and general CRC optimization suggestions. The table on the last page of the app note is the key to understanding these implementations.

Jack Crenshaw's "Implementing CRCs" article in the January 1992 isue of *Embedded Systems Programming*. This may be difficult to find, but it explains CRC's in very clear and concise terms. Well worth the effort to obtain a copy.

A typical application would look like:

```
// Dallas iButton test vector.
uint8_t serno[] = { 0x02, 0x1c, 0xb8, 0x01, 0, 0, 0, 0xa2 };

int
checkcrc (void)
{
    uint8_t crc = 0, i;

    for (i = 0; i < sizeof serno / sizeof serno[0]; i++)
        crc = _crc_ibutton_update (crc, serno[i]);

    return crc; // must be 0
}
```

21.31.2 Function Documentation

21.31.2.1 _crc16_update()

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

Optimized CRC-16 calculation.

Polynomial: $x^{16} + x^{15} + x^2 + 1$ (0xa001)
Initial value: `0xffff`

This CRC is normally used in disk-drive controllers.

The following is the equivalent functionality written in C.

```
uint16_t
crc16_update (uint16_t crc, uint8_t a)
{
    crc ^= a;
    for (int i = 0; i < 8; ++i)
    {
        if (crc & 1)
            crc = (crc >> 1) ^ 0xA001;
        else
            crc = crc >> 1;
    }

    return crc;
}
```

21.31.2.2 _crc8_ccitt_update() static uint8_t _crc8_ccitt_update (uint8_t *__crc*, uint8_t *__data*) [inline], [static]

Optimized CRC-8-CCITT calculation.

Polynomial: $x^8 + x^2 + x + 1$ (0xE0)

For use with simple CRC-8
Initial value: 0x0

For use with CRC-8-ROHC
Initial value: 0xff
Reference: http://tools.ietf.org/html/rfc3095#section-5.9.1

For use with CRC-8-ATM/ITU
Initial value: 0xff
Final XOR value: 0x55
Reference: http://www.itu.int/rec/T-REC-I.432.1-199902-I/en

The C equivalent has been originally written by Dave Hylands. Assembly code is based on _crc_ibutton_update optimization.

The following is the equivalent functionality written in C.

```
uint8_t
_crc8_ccitt_update (uint8_t inCrc, uint8_t inData)
{
    uint8_t data = inCrc ^ inData;

    for (int i = 0; i < 8; i++)
    {
        if ((data & 0x80) != 0)
        {
            data «= 1;
            data ^= 0x07;
        }
        else
        {
            data «= 1;
        }
    }
    return data;
}
```

21.31.2.3 _crc_ccitt_update() static uint16_t _crc_ccitt_update (uint16_t *__crc*, uint8_t *__data*) [inline], [static]

Optimized CRC-CCITT calculation.

Polynomial: $x^{16} + x^{12} + x^5 + 1$ (0x8408)
Initial value: 0xffff

This is the CRC used by PPP and IrDA.

See RFC1171 (PPP protocol) and IrDA IrLAP 1.1

Note

Although the CCITT polynomial is the same as that used by the Xmodem protocol, they are quite different. The difference is in how the bits are shifted through the alorgithm. Xmodem shifts the MSB of the CRC and the input first, while CCITT shifts the LSB of the CRC and the input first.

The following is the equivalent functionality written in C.

```
uint16_t
crc_ccitt_update (uint16_t crc, uint8_t data)
{
    data ^= lo8 (crc);
    data ^= data « 4;

    return ((((uint16_t)data « 8) | hi8 (crc)) ^ (uint8_t)(data » 4)
            ^ ((uint16_t)data « 3));
}
```

21.31.2.4 _crc_ibutton_update()

static uint8_t _crc_ibutton_update (
uint8_t *__crc*,
uint8_t *__data*) [inline], [static]

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

Polynomial: $x^8 + x^5 + x^4 + 1$ (0x8C)
Initial value: `0x0`

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

The following is the equivalent functionality written in C.

```
uint8_t
_crc_ibutton_update (uint8_t crc, uint8_t data)
{
    crc = crc ^ data;
    for (uint8_t i = 0; i < 8; i++)
    {
        if (crc & 0x01)
            crc = (crc » 1) ^ 0x8C;
        else
            crc »= 1;
    }

    return crc;
}
```

21.31.2.5 _crc_xmodem_update()

static uint16_t _crc_xmodem_update (
uint16_t *__crc*,
uint8_t *__data*) [inline], [static]

Optimized CRC-XMODEM calculation.

Polynomial: $x^{16} + x^{12} + x^5 + 1$ (0x1021)
Initial value: `0x0`

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

The following is the equivalent functionality written in C.

```
uint16_t
crc_xmodem_update (uint16_t crc, uint8_t data)
{
    crc = crc ^ ((uint16_t)data « 8);
    for (int i = 0; i < 8; i++)
    {
        if (crc & 0x8000)
            crc = (crc « 1) ^ 0x1021;
        else
            crc «= 1;
    }

    return crc;
}
```

21.32 <util/delay_basic.h>: Basic busy-wait delay loops

Functions

- void _delay_loop_1 (uint8_t __count)
- void _delay_loop_2 (uint16_t __count)

21.32.1 Detailed Description

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

The functions in this header file implement simple delay loops that perform a busy-waiting. They are typically used to facilitate short delays in the program execution. They are implemented as count-down loops with a well-known CPU cycle count per loop iteration. As such, no other processing can occur simultaneously. It should be kept in mind that the functions described here do not disable interrupts.

In general, for long delays, the use of hardware timers is much preferrable, as they free the CPU, and allow for concurrent processing of other events while the timer is running. However, in particular for very short delays, the overhead of setting up a hardware timer is too much compared to the overall delay time.

Two inline functions are provided for the actual delay algorithms.

21.32.2 Function Documentation

21.32.2.1 _delay_loop_1()

```
void _delay_loop_1 (
        uint8_t __count )
```

Delay loop using an 8-bit counter `__count`, so up to 256 iterations are possible. (The value 256 would have to be passed as 0.) The loop executes three CPU cycles per iteration, not including the overhead the compiler needs to setup the counter register.

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

21.32.2.2 _delay_loop_2()

```
void _delay_loop_2 (
        uint16_t __count )
```

Delay loop using a 16-bit counter `__count`, so up to 65536 iterations are possible. (The value 65536 would have to be passed as 0.) The loop executes four CPU cycles per iteration, not including the overhead the compiler requires to setup the counter register pair.

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

21.33 <util/eu_dst.h>: Daylight Saving function for the European Union.

Functions

- int eu_dst (const time_t *timer, int32_t *z)

21.33.1 Detailed Description

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

Daylight Saving Time for the European Union

21.33.2 Function Documentation

21.33.2.1 eu_dst()

```
int eu_dst (
    const time_t * timer,
    int32_t * z )
```

To utilize this function, call

```
set_dst(eu_dst);
```

Given the time stamp and time zone parameters provided, the Daylight Saving function must return a value appropriate for the tm structures' tm_isdst element. That is:

- `0` : If Daylight Saving is not in effect.
- `-1` : If it cannot be determined if Daylight Saving is in effect.
- A positive integer: Represents the number of seconds a clock is advanced for Daylight Saving. This will typically be ONE_HOUR.

Daylight Saving 'rules' are subject to frequent change. For production applications it is recommended to write your own DST function, which uses 'rules' obtained from, and modifiable by, the end user (perhaps stored in EEPROM).

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

Functions

- static uint8_t parity_even_bit (uint8_t __val)

21.34.1 Detailed Description

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

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

21.34.2 Function Documentation

21.34.2.1 parity_even_bit()

```
uint8_t parity_even_bit (
    uint8_t val ) [inline], [static]
```

Returns

1 if `val` has an odd number of bits set, and 0 otherwise.

21.35 <util/setbaud.h>: Helper macros for baud rate calculations

Macros

- #define BAUD_TOL 2
- #define UBRR_VALUE
- #define UBRRL_VALUE
- #define UBRRH_VALUE
- #define USE_2X 0

21.35.1 Detailed Description

```
#define F_CPU 11059200
#define BAUD 38400
#include <util/setbaud.h>
```

This header file requires that on entry values are already defined for F_CPU and BAUD. In addition, the macro BAUD_TOL will define the baud rate tolerance (in percent) that is acceptable during the calculations. The value of BAUD_TOL will default to 2 %.

This header file defines macros suitable to setup the UART baud rate prescaler registers of an AVR. All calculations are done using the C preprocessor. Including this header file causes no other side effects so it is possible to include this file more than once (supposedly, with different values for the BAUD parameter), possibly even within the same function.

Assuming that the requested BAUD is valid for the given F_CPU then the macro UBRR_VALUE is set to the required prescaler value. Two additional macros are provided for the low and high bytes of the prescaler, respectively↵ : UBRRL_VALUE is set to the lower byte of the UBRR_VALUE and UBRRH_VALUE is set to the upper byte. An additional macro USE_2X will be defined. Its value is set to 1 if the desired BAUD rate within the given tolerance could only be achieved by setting the U2X bit in the UART configuration. It will be defined to 0 if U2X is not needed.

Example usage:

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

#define F_CPU 4000000

static void
uart_9600(void)
{
#define BAUD 9600
#include <util/setbaud.h>
UBRRH = UBRRH_VALUE;
UBRRL = UBRRL_VALUE;
#if USE_2X
UCSRA |= (1 << U2X);
#else
UCSRA &= ~(1 << U2X);
#endif
}

static void
uart_38400(void)
{
#undef BAUD  // avoid compiler warning
#define BAUD 38400
#include <util/setbaud.h>
UBRRH = UBRRH_VALUE;
UBRRL = UBRRL_VALUE;
#if USE_2X
UCSRA |= (1 << U2X);
#else
UCSRA &= ~(1 << U2X);
#endif
}
```

In this example, two functions are defined to setup the UART to run at 9600 Bd, and 38400 Bd, respectively. Using a CPU clock of 4 MHz, 9600 Bd can be achieved with an acceptable tolerance without setting U2X (prescaler 25), while 38400 Bd require U2X to be set (prescaler 12).

21.35.2 Macro Definition Documentation

21.35.2.1 BAUD_TOL `#define BAUD_TOL 2`

Input and output macro for <util/setbaud.h>

Define the acceptable baud rate tolerance in percent. If not set on entry, it will be set to its default value of 2.

21.35.2.2 UBRR_VALUE `#define UBRR_VALUE`

Output macro from <util/setbaud.h>

Contains the calculated baud rate prescaler value for the UBRR register.

21.35.2.3 UBRRH_VALUE `#define UBRRH_VALUE`

Output macro from <util/setbaud.h>

Contains the upper byte of the calculated prescaler value (UBRR_VALUE).

21.35.2.4 UBRRL_VALUE `#define UBRRL_VALUE`

Output macro from <util/setbaud.h>

Contains the lower byte of the calculated prescaler value (UBRR_VALUE).

21.35.2.5 USE_2X `#define USE_2X 0`

Output macro from <util/setbaud.h>

Contains the value 1 if the desired baud rate tolerance could only be achieved by setting the U2X bit in the UART configuration. Contains 0 otherwise.

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

TWSR values

Mnemonics:
TW_MT_xxx - master transmitter
TW_MR_xxx - master receiver
TW_ST_xxx - slave transmitter
TW_SR_xxx - slave receiver

- #define TW_START 0x08
- #define TW_REP_START 0x10
- #define TW_MT_SLA_ACK 0x18
- #define TW_MT_SLA_NACK 0x20
- #define TW_MT_DATA_ACK 0x28
- #define TW_MT_DATA_NACK 0x30
- #define TW_MT_ARB_LOST 0x38
- #define TW_MR_ARB_LOST 0x38
- #define TW_MR_SLA_ACK 0x40
- #define TW_MR_SLA_NACK 0x48
- #define TW_MR_DATA_ACK 0x50
- #define TW_MR_DATA_NACK 0x58
- #define TW_ST_SLA_ACK 0xA8
- #define TW_ST_ARB_LOST_SLA_ACK 0xB0
- #define TW_ST_DATA_ACK 0xB8
- #define TW_ST_DATA_NACK 0xC0
- #define TW_ST_LAST_DATA 0xC8
- #define TW_SR_SLA_ACK 0x60
- #define TW_SR_ARB_LOST_SLA_ACK 0x68
- #define TW_SR_GCALL_ACK 0x70
- #define TW_SR_ARB_LOST_GCALL_ACK 0x78
- #define TW_SR_DATA_ACK 0x80
- #define TW_SR_DATA_NACK 0x88
- #define TW_SR_GCALL_DATA_ACK 0x90
- #define TW_SR_GCALL_DATA_NACK 0x98
- #define TW_SR_STOP 0xA0
- #define TW_NO_INFO 0xF8
- #define TW_BUS_ERROR 0x00
- #define TW_STATUS_MASK
- #define TW_STATUS (TWSR & TW_STATUS_MASK)

R/∼W bit in SLA+R/W address field.

- #define TW_READ 1
- #define TW_WRITE 0

21.36.1 Detailed Description

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

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

21.36.2 Macro Definition Documentation

21.36.2.1 TW_BUS_ERROR `#define TW_BUS_ERROR 0x00`

illegal start or stop condition

21.36.2.2 TW_MR_ARB_LOST `#define TW_MR_ARB_LOST 0x38`

arbitration lost in SLA+R or NACK

21.36.2.3 TW_MR_DATA_ACK `#define TW_MR_DATA_ACK 0x50`

data received, ACK returned

21.36.2.4 TW_MR_DATA_NACK `#define TW_MR_DATA_NACK 0x58`

data received, NACK returned

21.36.2.5 TW_MR_SLA_ACK `#define TW_MR_SLA_ACK 0x40`

SLA+R transmitted, ACK received

21.36.2.6 TW_MR_SLA_NACK `#define TW_MR_SLA_NACK 0x48`

SLA+R transmitted, NACK received

21.36.2.7 TW_MT_ARB_LOST `#define TW_MT_ARB_LOST 0x38`

arbitration lost in SLA+W or data

21.36.2.8 TW_MT_DATA_ACK `#define TW_MT_DATA_ACK 0x28`

data transmitted, ACK received

21.36.2.9 TW_MT_DATA_NACK `#define TW_MT_DATA_NACK 0x30`

data transmitted, NACK received

21.36.2.10 TW_MT_SLA_ACK `#define TW_MT_SLA_ACK 0x18`

SLA+W transmitted, ACK received

21.36.2.11 TW_MT_SLA_NACK `#define TW_MT_SLA_NACK 0x20`

SLA+W transmitted, NACK received

21.36.2.12 TW_NO_INFO `#define TW_NO_INFO 0xF8`

no state information available

21.36.2.13 TW_READ `#define TW_READ 1`

SLA+R address

21.36.2.14 TW_REP_START `#define TW_REP_START 0x10`

repeated start condition transmitted

21.36.2.15 TW_SR_ARB_LOST_GCALL_ACK `#define TW_SR_ARB_LOST_GCALL_ACK 0x78`

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

21.36.2.16 TW_SR_ARB_LOST_SLA_ACK `#define TW_SR_ARB_LOST_SLA_ACK 0x68`

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

21.36.2.17 TW_SR_DATA_ACK `#define TW_SR_DATA_ACK 0x80`

data received, ACK returned

21.36.2.18 TW_SR_DATA_NACK `#define TW_SR_DATA_NACK 0x88`

data received, NACK returned

21.36.2.19 TW_SR_GCALL_ACK `#define TW_SR_GCALL_ACK 0x70`

general call received, ACK returned

21.36.2.20 TW_SR_GCALL_DATA_ACK `#define TW_SR_GCALL_DATA_ACK 0x90`

general call data received, ACK returned

21.36.2.21 TW_SR_GCALL_DATA_NACK `#define TW_SR_GCALL_DATA_NACK 0x98`

general call data received, NACK returned

21.36.2.22 TW_SR_SLA_ACK `#define TW_SR_SLA_ACK 0x60`

SLA+W received, ACK returned

21.36.2.23 TW_SR_STOP `#define TW_SR_STOP 0xA0`

stop or repeated start condition received while selected

21.36.2.24 TW_ST_ARB_LOST_SLA_ACK `#define TW_ST_ARB_LOST_SLA_ACK 0xB0`

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

21.36.2.25 TW_ST_DATA_ACK `#define TW_ST_DATA_ACK 0xB8`

data transmitted, ACK received

21.36.2.26 TW_ST_DATA_NACK `#define TW_ST_DATA_NACK 0xC0`

data transmitted, NACK received

21.36.2.27 TW_ST_LAST_DATA `#define TW_ST_LAST_DATA 0xC8`

last data byte transmitted, ACK received

21.36.2.28 TW_ST_SLA_ACK `#define TW_ST_SLA_ACK 0xA8`

SLA+R received, ACK returned

21.36.2.29 TW_START `#define TW_START 0x08`

start condition transmitted

21.36.2.30 TW_STATUS `#define TW_STATUS (TWSR & TW_STATUS_MASK)`

TWSR, masked by TW_STATUS_MASK

21.36.2.31 TW_STATUS_MASK `#define TW_STATUS_MASK`

Value:

```
(_BV(TWS7)|_BV(TWS6)|_BV(TWS5)|_BV(TWS4)|\
_BV(TWS3))
```

The lower 3 bits of TWSR are reserved on the ATmega163. The 2 LSB carry the prescaler bits on the newer ATmegas.

21.36.2.32 TW_WRITE `#define TW_WRITE 0`

SLA+W address

21.37 <util/usa_dst.h>: Daylight Saving function for the USA.

Functions

- int usa_dst (const time_t *timer, int32_t *z)

21.37.1 Detailed Description

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

Daylight Saving function for the USA.

21.37.2 Function Documentation

21.37.2.1 usa_dst()

```
int usa_dst (
    const time_t * timer,
    int32_t * z )
```

To utilize this function, call

```
set_dst(usa_dst);
```

Given the time stamp and time zone parameters provided, the Daylight Saving function must return a value appropriate for the tm structures' tm_isdst element. That is:

- `0` : If Daylight Saving is not in effect.
- `-1` : If it cannot be determined if Daylight Saving is in effect.
- A positive integer : Represents the number of seconds a clock is advanced for Daylight Saving. This will typically be ONE_HOUR.

Daylight Saving 'rules' are subject to frequent change. For production applications it is recommended to write your own DST function, which uses 'rules' obtained from, and modifiable by, the end user (perhaps stored in EEPROM).

21.38 <compat/deprecated.h>: Deprecated items

Allowing specific system-wide interrupts

In addition to globally enabling interrupts, each device's particular interrupt needs to be enabled separately if interrupts for this device are desired. While some devices maintain their interrupt enable bit inside the device's register set, external and timer interrupts have system-wide configuration registers.

Example:
```
// Enable timer 1 overflow interrupts.
timer_enable_int(_BV(TOIE1));

// Do some work...

// Disable all timer interrupts.
timer_enable_int(0);
```

Note

> Be careful when you use these functions. If you already have a different interrupt enabled, you could inadvertantly disable it by enabling another intterupt.

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

Obsolete IO macros

Back in a time when AVR-GCC and AVR-LibC could not handle IO port access in the direct assignment form as they are handled now, all IO port access had to be done through specific macros that eventually resulted in inline assembly instructions performing the desired action.

These macros became obsolete, as reading and writing IO ports can be done by simply using the IO port name in an expression, and all bit manipulation (including those on IO ports) can be done using generic C bit manipulation operators.

The macros in this group simulate the historical behaviour. While they are supposed to be applied to IO ports, the emulation actually uses standard C methods, so they could be applied to arbitrary memory locations as well.

- #define inp(port) (port)
- #define outp(val, port) (port) = (val)
- #define inb(port) (port)
- #define outb(port, val) (port) = (val)
- #define sbi(port, bit) (port) |= (1 << (bit))
- #define cbi(port, bit) (port) &= ~(1 << (bit))

21.38.1 Detailed Description

This header file contains several items that used to be available in previous versions of this library, but have eventually been deprecated over time.
```
#include <compat/deprecated.h>
```

These items are supplied within that header file for backward compatibility reasons only, so old source code that has been written for previous library versions could easily be maintained until its end-of-life. Use of any of these items in new code is strongly discouraged.

21.38.2 Macro Definition Documentation

21.38.2.1 cbi

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

Deprecated

Clear `bit` in IO port `port`.

21.38.2.2 enable_external_int

```
#define enable_external_int(
        mask ) (__EICR = mask)
```

Deprecated

This macro gives access to the `GIMSK` register (or `EIMSK` register if using an AVR Mega device or `GICR` register for others). Although this macro is essentially the same as assigning to the register, it does adapt slightly to the type of device being used. This macro is unavailable if none of the registers listed above are defined.

21.38.2.3 inb

```
#define inb(
        port ) (port)
```

Deprecated

Read a value from an IO port `port`.

21.38.2.4 inp

```
#define inp(
        port ) (port)
```

Deprecated

Read a value from an IO port `port`.

21.38.2.5 INTERRUPT

```
#define INTERRUPT(
        signame )
```

Value:

```
void signame (void) __attribute__ ((__interrupt__,__INTR_ATTRS));    \
void signame (void)
```

Deprecated

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

As this macro has been used by too many unsuspecting people in the past, it has been deprecated, and will be removed in a future version of the library. Users who want to legitimately re-enable interrupts in their interrupt handlers as quickly as possible are encouraged to explicitly declare their handlers as described above.

21.38.2.6 outb

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

Deprecated

Write `val` to IO port `port`.

21.38.2.7 outp

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

Deprecated

Write `val` to IO port `port`.

21.38.2.8 sbi

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

Deprecated

Set `bit` in IO port `port`.

21.38.3 Function Documentation

21.38.3.1 timer_enable_int()

```
static void timer_enable_int (
    unsigned char ints ) [inline], [static]
```

Deprecated

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

21.39 <compat/ina90.h>: Compatibility with IAR EWB 3.x

```
#include <compat/ina90.h>
```

This is an attempt to provide some compatibility with header files that come with IAR C, to make porting applications between different compilers easier. No 100% compatibility though.

Note

For actual documentation, please see the IAR manual.

21.40 Demo projects

Modules

- Combining C and assembly source files
- A simple project
- A more sophisticated project
- Using the standard IO facilities
- Example using the two-wire interface (TWI)

21.40.1 Detailed Description

Various small demo projects are provided to illustrate several aspects of using the opensource utilities for the AVR controller series. It should be kept in mind that these demos serve mainly educational purposes, and are normally not directly suitable for use in any production environment. Usually, they have been kept as simple as sufficient to demonstrate one particular feature.

The simple project is somewhat like the "Hello world!" application for a microcontroller, about the most simple project that can be done. It is explained in good detail, to allow the reader to understand the basic concepts behind using the tools on an AVR microcontroller.

The more sophisticated demo project builds on top of that simple project, and adds some controls to it. It touches a number of AVR-LibC's basic concepts on its way.

A comprehensive example on using the standard IO facilities intends to explain that complex topic, using a practical microcontroller peripheral setup with one RS-232 connection, and an HD44780-compatible industry-standard LCD display.

The Example using the two-wire interface (TWI) project explains the use of the two-wire hardware interface (also known as "I2C") that is present on many AVR controllers.

Finally, the Combining C and assembly source files demo shows how C and assembly language source files can collaborate within one project. While the overall project is managed by a C program part for easy maintenance, time-critical parts are written directly in manually optimized assembly language for shortest execution times possible. Naturally, this kind of project is very closely tied to the hardware design, thus it is custom-tailored to a particular controller type and peripheral setup. As an alternative to the assembly-language solution, this project also offers a C-only implementation (deploying the exact same peripheral setup) based on a more sophisticated (and thus more expensive) but pin-compatible controller.

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

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

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

21.41 Combining C and assembly source files

For time- or space-critical applications, it can often be desirable to combine C code (for easy maintenance) and assembly code (for maximal speed or minimal code size) together. This demo provides an example of how to do that.

The objective of the demo is to decode radio-controlled model PWM signals, and control an output PWM based on the current input signal's value. The incoming PWM pulses follow a standard encoding scheme where a pulse width of 920 microseconds denotes one end of the scale (represented as 0 % pulse width on output), and 2120 microseconds mark the other end (100 % output PWM). Normally, multiple channels would be encoded that way in subsequent pulses, followed by a larger gap, so the entire frame will repeat each 14 through 20 ms, but this is ignored for the purpose of the demo, so only a single input PWM channel is assumed.

The basic challenge is to use the cheapest controller available for the task, an ATtiny13 that has only a single timer channel. As this timer channel is required to run the outgoing PWM signal generation, the incoming PWM decoding had to be adjusted to the constraints set by the outgoing PWM.

As PWM generation toggles the counting direction of timer 0 between up and down after each 256 timer cycles, the current time cannot be deduced by reading TCNT0 only, but the current counting direction of the timer needs to be considered as well. This requires servicing interrupts whenever the timer hits *TOP* (255) and *BOTTOM* (0) to learn about each change of the counting direction. For PWM generation, it is usually desired to run it at the highest possible speed so filtering the PWM frequency from the modulated output signal is made easy. Thus, the PWM timer runs at full CPU speed. This causes the overflow and compare match interrupts to be triggered each 256 CPU clocks, so they must run with the minimal number of processor cycles possible in order to not impose a too high CPU load by these interrupt service routines. This is the main reason to implement the entire interrupt handling in fine-tuned assembly code rather than in C.

In order to verify parts of the algorithm, and the underlying hardware, the demo has been set up in a way so the pin-compatible but more expensive ATtiny45 (or its siblings ATtiny25 and ATtiny85) could be used as well. In that case, no separate assembly code is required, as two timer channels are avaible.

21.41.1 Hardware setup

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

The outgoing PWM is generated through OC0B of timer channel 0 (PB1). For demonstration purposes, a LED should be connected to that pin (like, one of the LEDs of an STK500).

The controllers run on their internal calibrated RC oscillators, 1.2 MHz on the ATtiny13, and 1.0 MHz on the ATtiny45.

21.41.2 A code walkthrough

21.41.2.1 asmdemo.c After the usual include files, two variables are defined. The first one, `pwm_incoming` is used to communicate the most recent pulse width detected by the incoming PWM decoder up to the main loop.

The second variable actually only constitutes of a single bit, `intbits.pwm_received`. This bit will be set whenever the incoming PWM decoder has updated `pwm_incoming`.

Both variables are marked *volatile* to ensure their readers will always pick up an updated value, as both variables will be set by interrupt service routines.

The function `ioinit()` initializes the microcontroller peripheral devices. In particular, it starts timer 0 to generate the outgoing PWM signal on OC0B. Setting OCR0A to 255 (which is the *TOP* value of timer 0) is used to generate a timer 0 overflow A interrupt on the ATtiny13. This interrupt is used to inform the incoming PWM decoder that the counting direction of channel 0 is just changing from up to down. Likewise, an overflow interrupt will be generated whenever the countdown reached *BOTTOM* (value 0), where the counter will again alter its counting direction to upwards. This information is needed in order to know whether the current counter value of `TCNT0` is to be evaluated from bottom or top.

Further, `ioinit()` activates the pin-change interrupt `PCINT0` on any edge of PB4. Finally, PB1 (OC0B) will be activated as an output pin, and global interrupts are being enabled.

In the ATtiny45 setup, the C code contains an ISR for `PCINT0`. At each pin-change interrupt, it will first be analyzed whether the interrupt was caused by a rising or a falling edge. In case of the rising edge, timer 1 will be started with a prescaler of 16 after clearing the current timer value. Then, at the falling edge, the current timer value will be recorded (and timer 1 stopped), the pin-change interrupt will be suspended, and the upper layer will be notified that the incoming PWM measurement data is available.

Function `main()` first initializes the hardware by calling `ioinit()`, and then waits until some incoming PWM value is available. If it is, the output PWM will be adjusted by computing the relative value of the incoming PWM. Finally, the pin-change interrupt is re-enabled, and the CPU is put to sleep.

21.41.2.2 project.h In order for the interrupt service routines to be as fast as possible, some of the CPU registers are set aside completely for use by these routines, so the compiler would not use them for C code. This is arranged for in `project.h`.

The file is divided into one section that will be used by the assembly source code, and another one to be used by C code. The assembly part is distinguished by the preprocessing macro `__ASSEMBLER__` (which will be automatically set by the compiler front-end when preprocessing an assembly-language file), and it contains just macros that give symbolic names to a number of CPU registers. The preprocessor will then replace the symbolic names by their right-hand side definitions before calling the assembler.

In C code, the compiler needs to see variable declarations for these objects. This is done by using declarations that bind a variable permanently to a CPU register (see How to permanently bind a variable to a register?). Even in case the C code never has a need to access these variables, declaring the register binding that way causes the compiler to not use these registers in C code at all.

The `flags` variable needs to be in the range of r16 through r31 as it is the target of a *load immediate* (or `SER`) instruction that is not applicable to the entire register file.

21.41.2.3 isrs.S This file is a preprocessed assembly source file. The C preprocessor will be run by the compiler front-end first, resolving all `#include`, `#define` etc. directives. The resulting program text will then be passed on to the assembler.

As the C preprocessor strips all C-style comments, preprocessed assembly source files can have both, C-style (`/* ... */`, `// ...`) as well as assembly-style (`; ...`) comments.

At the top, the IO register definition file `avr/io.h` and the project declaration file `project.h` are included. The remainder of the file is conditionally assembled only if the target MCU type is an ATtiny13, so it will be completely ignored for the ATtiny45 option.

Next are the two interrupt service routines for timer 0 compare A match (timer 0 hits *TOP*, as OCR0A is set to 255) and timer 0 overflow (timer 0 hits *BOTTOM*). As discussed above, these are kept as short as possible. They only save `SREG` (as the flags will be modified by the `INC` instruction), increment the `counter_hi` variable which forms the high part of the current time counter (the low part is formed by querying `TCNT0` directly), and clear or set the variable `flags`, respectively, in order to note the current counting direction. The `RETI` instruction terminates these interrupt service routines. Total cycle count is 8 CPU cycles, so together with the 4 CPU cycles needed for interrupt setup, and the 2 cycles for the RJMP from the interrupt vector to the handler, these routines will require 14 out of each 256 CPU cycles, or about 5 % of the overall CPU time.

The pin-change interrupt `PCINT0` will be handled in the final part of this file. The basic algorithm is to quickly evaluate the current system time by fetching the current timer value of `TCNT0`, and combining it with the overflow part in `counter_hi`. If the counter is currently counting down rather than up, the value fetched from `TCNT0` must be negated. Finally, if this pin-change interrupt was triggered by a rising edge, the time computed will be recorded as the start time only. Then, at the falling edge, this start time will be subracted from the current time to compute the actual pulse width seen (left in `pwm_incoming`), and the upper layers are informed of the new value by setting bit 0 in the `intbits` flags. At the same time, this pin-change interrupt will be disabled so no new measurement can be performed until the upper layer had a chance to process the current value.

21.41.3 The source code

The source code is installed under

`$prefix/share/doc/avr-libc/examples/asmdemo/`,

where `$prefix` is a configuration option. For Unix systems, it is usually set to either `/usr` or `/usr/local`.

21.42 A simple project

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

21.42.1 The Project

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

Note

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

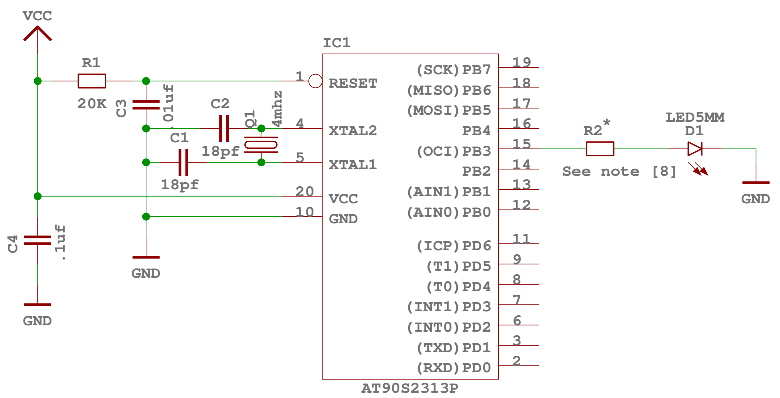

Figure 4 Schematic of circuit for demo project

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

Note [1]:

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

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

Note [2]:

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

Note [3]:

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

Note [4]:

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

Note [5]:

Here's where the newly computed value is loaded into the `PWM` register. Since we are in an interrupt routine, it is safe to use a 16-bit assignment to the register. Outside of an interrupt, the assignment should only be performed with interrupts disabled if there's a chance that an interrupt routine could also access this register (or another register that uses `TEMP`), see the appropriate FAQ entry.

Note [6]:

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

Note [7]:

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

Note [8]:

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

21.42.2 The Source Code

```
/*
 * ----------------------------------------------------------------------------
 * "THE BEER-WARE LICENSE" (Revision 42):
 * <joerg@FreeBSD.ORG> wrote this file.  As long as you retain this notice you
 * can do whatever you want with this stuff. If we meet some day, and you think
 * this stuff is worth it, you can buy me a beer in return.        Joerg Wunsch
 * ----------------------------------------------------------------------------
 *
 * Simple AVR demonstration.  Controls a LED that can be directly
 * connected from OC1/OC1A to GND.  The brightness of the LED is
 * controlled with the PWM.  After each period of the PWM, the PWM
 * value is either incremented or decremented, that's all.
 *
 * $Id$
 */

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

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

enum { UP, DOWN };

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

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

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

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

void
ioinit (void)           /* Note [6] */
{
    /* Timer 1 is 10-bit PWM (8-bit PWM on some ATtinys). */
    TCCR1A = TIMER1_PWM_INIT;
    /*
     * Start timer 1.
     *
     * NB: TCCR1A and TCCR1B could actually be the same register, so
     * take care to not clobber it.
     */
    TCCR1B |= TIMER1_CLOCKSOURCE;
    /*
     * Run any device-dependent timer 1 setup hook if present.
     */
#if defined(TIMER1_SETUP_HOOK)
    TIMER1_SETUP_HOOK();
#endif

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

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

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

int
main (void)
{

    ioinit ();

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

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

```
        sleep_mode();

    return (0);
}
```

21.42.3 Compiling and Linking

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

```
$ avr-gcc -g -Os -mmcu=atmega8 -c demo.c
```

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

```
$ avr-gcc -g -mmcu=atmega8 -o demo.elf demo.o
```

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

21.42.4 Examining the Object File

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

For instance, to get a feel of the application's size, the -h option can be used. The output of this option shows how much space is used in each of the sections (the .stab and .stabstr sections hold the debugging information and won't make it into the ROM file).

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

```
$ avr-objdump -h -S demo.elf > demo.lst
```

Here's the output as saved in the demo.lst file:

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

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         000000dc  00000000  00000000  00000094  2**1
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .data         00000000  00800060  000000dc  00000170  2**0
                  CONTENTS, ALLOC, LOAD, DATA
  2 .bss          00000003  00800060  00800060  00000170  2**0
                  ALLOC
  3 .stab         00000234  00000000  00000000  00000170  2**2
                  CONTENTS, READONLY, DEBUGGING
  4 .stabstr      0000015e  00000000  00000000  000003a4  2**0
                  CONTENTS, READONLY, DEBUGGING
  5 .comment      00000012  00000000  00000000  00000502  2**0
                  CONTENTS, READONLY
  6 .note.gnu.avr.deviceinfo 0000003c  00000000  00000000  00000514  2**2
                  CONTENTS, READONLY
  7 .debug_info   0000048c  00000000  00000000  00000550  2**0
                  CONTENTS, READONLY, DEBUGGING
  8 .debug_abbrev 0000044e  00000000  00000000  000009dc  2**0
                  CONTENTS, READONLY, DEBUGGING
  9 .debug_line   0000001d  00000000  00000000  00000e2a  2**0
                  CONTENTS, READONLY, DEBUGGING
 10 .debug_str    0000017a  00000000  00000000  00000e47  2**0
                  CONTENTS, READONLY, DEBUGGING

Disassembly of section .text:

00000000 <__vectors>:
   0: 12 c0        rjmp .+36      ; 0x26 <__ctors_end>
   2: 5e c0        rjmp .+188     ; 0xc0 <__bad_interrupt>
   4: 5d c0        rjmp .+186     ; 0xc0 <__bad_interrupt>
   6: 5c c0        rjmp .+184     ; 0xc0 <__bad_interrupt>
   8: 5b c0        rjmp .+182     ; 0xc0 <__bad_interrupt>
   a: 5a c0        rjmp .+180     ; 0xc0 <__bad_interrupt>
   c: 59 c0        rjmp .+178     ; 0xc0 <__bad_interrupt>
   e: 58 c0        rjmp .+176     ; 0xc0 <__bad_interrupt>
  10: 1a c0        rjmp .+52      ; 0x46 <__vector_8>
  12: 56 c0        rjmp .+172     ; 0xc0 <__bad_interrupt>
  14: 55 c0        rjmp .+170     ; 0xc0 <__bad_interrupt>
  16: 54 c0        rjmp .+168     ; 0xc0 <__bad_interrupt>
  18: 53 c0        rjmp .+166     ; 0xc0 <__bad_interrupt>
  1a: 52 c0        rjmp .+164     ; 0xc0 <__bad_interrupt>
  1c: 51 c0        rjmp .+162     ; 0xc0 <__bad_interrupt>
  1e: 50 c0        rjmp .+160     ; 0xc0 <__bad_interrupt>
  20: 4f c0        rjmp .+158     ; 0xc0 <__bad_interrupt>
  22: 4e c0        rjmp .+156     ; 0xc0 <__bad_interrupt>
  24: 4d c0        rjmp .+154     ; 0xc0 <__bad_interrupt>

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

00000032 <__do_clear_bss>:
  32: 20 e0        ldi r18, 0x00 ; 0
  34: a0 e6        ldi r26, 0x60 ; 96
  36: b0 e0        ldi r27, 0x00 ; 0
  38: 01 c0        rjmp .+2       ; 0x3c <.do_clear_bss_start>

0000003a <.do_clear_bss_loop>:
  3a: 1d 92        st X+, r1

0000003c <.do_clear_bss_start>:
  3c: a3 36        cpi r26, 0x63 ; 99
  3e: b2 07        cpc r27, r18
  40: e1 f7        brne .-8       ; 0x3a <.do_clear_bss_loop>
  42: 3f d0        rcall .+126     ; 0xc2 <main>
  44: 47 c0        rjmp .+142     ; 0xd4 <exit>
```

```
00000046 <__vector_8>:
#include "iocompat.h" /* Note [1] */

enum { UP, DOWN };

ISR (TIMER1_OVF_vect) /* Note [2] */
{
  46: 1f 92         push r1
  48: 1f b6         in r1, 0x3f ; 63
  4a: 1f 92         push r1
  4c: 11 24         eor r1, r1
  4e: 2f 93         push r18
  50: 8f 93         push r24
  52: 9f 93         push r25
    static uint16_t pwm; /* Note [3] */
    static uint8_t direction;

    switch (direction) /* Note [4] */
  54: 20 91 62 00  lds r18, 0x0062 ; 0x800062 <direction.1>
    {
        case UP:
            if (++pwm == TIMER1_TOP)
  58: 80 91 60 00  lds r24, 0x0060 ; 0x800060 <__DATA_REGION_ORIGIN__>
  5c: 90 91 61 00  lds r25, 0x0061 ; 0x800061 <__DATA_REGION_ORIGIN__+0x1>
    switch (direction) /* Note [4] */
  60: 22 23         and r18, r18
  62: a1 f0         breq .+40      ; 0x8c <__vector_8+0x46>
  64: 21 30         cpi r18, 0x01 ; 1
  66: 49 f4         brne .+18      ; 0x7a <__vector_8+0x34>
                direction = DOWN;
            break;

        case DOWN:
            if (--pwm == 0)
  68: 01 97         sbiw r24, 0x01 ; 1
  6a: 90 93 61 00  sts 0x0061, r25 ; 0x800061 <__DATA_REGION_ORIGIN__+0x1>
  6e: 80 93 60 00  sts 0x0060, r24 ; 0x800060 <__DATA_REGION_ORIGIN__>
  72: 00 97         sbiw r24, 0x00 ; 0
  74: 11 f4         brne .+4       ; 0x7a <__vector_8+0x34>
                direction = UP;
  76: 10 92 62 00  sts 0x0062, r1 ; 0x800062 <direction.1>
            break;
    }

    OCR = pwm; /* Note [5] */
  7a: 9b bd         out 0x2b, r25 ; 43
  7c: 8a bd         out 0x2a, r24 ; 42
}
  7e: 9f 91         pop r25
  80: 8f 91         pop r24
  82: 2f 91         pop r18
  84: 1f 90         pop r1
  86: 1f be         out 0x3f, r1 ; 63
  88: 1f 90         pop r1
  8a: 18 95         reti
            if (++pwm == TIMER1_TOP)
  8c: 01 96         adiw r24, 0x01 ; 1
  8e: 90 93 61 00  sts 0x0061, r25 ; 0x800061 <__DATA_REGION_ORIGIN__+0x1>
  92: 80 93 60 00  sts 0x0060, r24 ; 0x800060 <__DATA_REGION_ORIGIN__>
  96: 8f 3f         cpi r24, 0xFF ; 255
  98: 23 e0         ldi r18, 0x03 ; 3
  9a: 92 07         cpc r25, r18
  9c: 71 f7         brne .-36      ; 0x7a <__vector_8+0x34>
                direction = DOWN;
  9e: 21 e0         ldi r18, 0x01 ; 1
  a0: 20 93 62 00  sts 0x0062, r18 ; 0x800062 <direction.1>
  a4: ea cf         rjmp .-44      ; 0x7a <__vector_8+0x34>

000000a6 <ioinit>:

void
ioinit (void) /* Note [6] */
```

```
{
    /* Timer 1 is 10-bit PWM (8-bit PWM on some ATtinys). */
    TCCR1A = TIMER1_PWM_INIT;
  a6: 83 e8        ldi r24, 0x83 ; 131
  a8: 8f bd        out 0x2f, r24 ; 47
     * Start timer 1.
     *
     * NB: TCCR1A and TCCR1B could actually be the same register, so
     * take care to not clobber it.
     */
    TCCR1B |= TIMER1_CLOCKSOURCE;
  aa: 8e b5        in r24, 0x2e ; 46
  ac: 81 60        ori r24, 0x01 ; 1
  ae: 8e bd        out 0x2e, r24 ; 46
#if defined(TIMER1_SETUP_HOOK)
    TIMER1_SETUP_HOOK();
#endif

    /* Set PWM value to 0. */
    OCR = 0;
  b0: 1b bc        out 0x2b, r1 ; 43
  b2: 1a bc        out 0x2a, r1 ; 42

    /* Enable OC1 as output. */
    DDROC = _BV (OC1);
  b4: 82 e0        ldi r24, 0x02 ; 2
  b6: 87 bb        out 0x17, r24 ; 23

    /* Enable timer 1 overflow interrupt. */
    TIMSK = _BV (TOIE1);
  b8: 84 e0        ldi r24, 0x04 ; 4
  ba: 89 bf        out 0x39, r24 ; 57
    sei ();
  bc: 78 94        sei
}
  be: 08 95        ret

000000c0 <__bad_interrupt>:
  c0: 9f cf        rjmp .-194     ; 0x0 <__vectors>

000000c2 <main>:

int
main (void)
{

    ioinit ();
  c2: f1 df        rcall .-30      ; 0xa6 <ioinit>

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

    for (;;) /* Note [7] */
        sleep_mode();
  c4: 85 b7        in r24, 0x35 ; 53
  c6: 80 68        ori r24, 0x80 ; 128
  c8: 85 bf        out 0x35, r24 ; 53
  ca: 88 95        sleep
  cc: 85 b7        in r24, 0x35 ; 53
  ce: 8f 77        andi r24, 0x7F ; 127
  d0: 85 bf        out 0x35, r24 ; 53
  d2: f8 cf        rjmp .-16      ; 0xc4 <main+0x2>

000000d4 <exit>:
  d4: f8 94        cli
  d6: 00 c0        rjmp .+0       ; 0xd8 <_exit>

000000d8 <_exit>:
  d8: f8 94        cli

000000da <__stop_program>:
  da: ff cf        rjmp .-2       ; 0xda <__stop_program>
```

21.42.5 Linker Map Files

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

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

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

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

.text           0x0000000000000000       0xdc
 *(.vectors)
 .vectors       0x0000000000000000       0x26 /home/joerg/src/avr-libc/avr/devices/atmega8/crtatmega8.o
                0x0000000000000000                __vectors
                0x0000000000000000                __vector_default
 *(.vectors)
 *(.progmem.gcc*)
                0x0000000000000026                . = ALIGN (0x2)
                0x0000000000000026                __trampolines_start = .
 *(.trampolines)
 .trampolines   0x0000000000000026        0x0 linker stubs
 *(.trampolines*)
                0x0000000000000026                __trampolines_end = .
 *libprintf_flt.a:*(.progmem.data)
 *libc.a:*(.progmem.data)
 *(.progmem.*)
                0x0000000000000026                . = ALIGN (0x2)
 *(.lowtext)
 *(.lowtext*)
                0x0000000000000026                __ctors_start = .
```

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

```
 *(.fini2)
 *(.fini2)
 *(.fini1)
 *(.fini1)
 *(.fini0)
 .fini0         0x00000000000000d8        0x4 /usr/local/lib/gcc/avr/11.2.0/avr4/libgcc.a(_exit.o)
 *(.fini0)
 *(.hightext)
 *(.hightext*)
 *(.progmemx.*)
                0x00000000000000dc                . = ALIGN (0x2)
 *(.jumptables)
 *(.jumptables*)
                0x00000000000000dc                _etext = .

.data           0x0000000000800060        0x0 load address 0x00000000000000dc
                [!provide]                        PROVIDE (__data_start = .)
 *(.data)
 .data          0x0000000000800060        0x0 demo.o
 .data          0x0000000000800060        0x0 /home/joerg/src/avr-libc/avr/lib/avr4/exit.o
 .data          0x0000000000800060        0x0 /home/joerg/src/avr-libc/avr/devices/atmega8/crtatmega8.o
 .data          0x0000000000800060        0x0 /usr/local/lib/gcc/avr/11.2.0/avr4/libgcc.a(_exit.o)
 .data          0x0000000000800060        0x0 /usr/local/lib/gcc/avr/11.2.0/avr4/libgcc.a(_clear_bss.o)
 *(.data*)
 *(.gnu.linkonce.d*)
 *(.rodata)
 *(.rodata*)
 *(.gnu.linkonce.r*)
                0x0000000000800060                . = ALIGN (0x2)
                0x0000000000800060                _edata = .
                [!provide]                        PROVIDE (__data_end = .)

.bss            0x0000000000800060        0x3
                0x0000000000800060                PROVIDE (__bss_start = .)
 *(.bss)
 .bss           0x0000000000800060        0x3 demo.o
 .bss           0x0000000000800063        0x0 /home/joerg/src/avr-libc/avr/lib/avr4/exit.o
 .bss           0x0000000000800063        0x0 /home/joerg/src/avr-libc/avr/devices/atmega8/crtatmega8.o
 .bss           0x0000000000800063        0x0 /usr/local/lib/gcc/avr/11.2.0/avr4/libgcc.a(_exit.o)
 .bss           0x0000000000800063        0x0 /usr/local/lib/gcc/avr/11.2.0/avr4/libgcc.a(_clear_bss.o)
 *(.bss*)
 *(COMMON)
                0x0000000000800063                PROVIDE (__bss_end = .)
```

```
                0x00000000000000dc                __data_load_start = LOADADDR (.data)
                0x00000000000000dc                __data_load_end = (__data_load_start + SIZEOF (.data))

.noinit         0x0000000000800063        0x0
                [!provide]                        PROVIDE (__noinit_start = .)
 *(.noinit*)
                [!provide]                        PROVIDE (__noinit_end = .)
                0x0000000000800063                _end = .
                [!provide]                        PROVIDE (__heap_start = .)

.eeprom         0x0000000000810000        0x0
 *(.eeprom*)
                0x0000000000810000                __eeprom_end = .
```

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

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

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

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

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

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

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

21.42.6 Generating Intel Hex Files

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

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

```
$ avr-objcopy -j .text -j .data -O ihex demo.elf demo.hex
```

The resulting `demo.hex` file contains:

```
:1000000012C05EC05DC05CC05BC05AC059C058C061
:100010001AC056C055C054C053C052C051C050C081
:100020004FC04EC04DC011241FBECFE5D4E0DEBF8F
:10003000CDBF20E0A0E6B0E001C01D92A336B2071C
:10004000E1F73FD047C01F921FB61F9211242F9394
:100050008F939F93209162008091600090916100046
:100060002223A1F0213049F40197909361008093FD
:100070006000009711F4109262009BBD8ABD9F91B1
:100080008F912F911F901FBE1F90189501969093EE
:10009000610080936000 8F3F23E0920771F721E0B9
:1000A00020936200EACF83E88FBD8EB581608EBD5C
:1000B0001BBC1ABC82E087BB84E089BF789408959A
:1000C0009FCFF1DF85B7806885BF889585B78F772B
:0C00D00085BFF8CFF89400C0F894FFCF73
:00000001FF
```

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

```
$ avr-objcopy -j .eeprom --change-section-lma .eeprom=0 -O ihex demo.elf demo_eeprom.hex
```

There is no `demo_eeprom.hex` file written, as that file would be empty.

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

21.42.7 Letting Make Build the Project

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

Note

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

```
PRG            = demo
OBJ            = demo.o
#MCU_TARGET     = at90s2313
#MCU_TARGET     = at90s2333
#MCU_TARGET     = at90s4414
#MCU_TARGET     = at90s4433
#MCU_TARGET     = at90s4434
#MCU_TARGET     = at90s8515
#MCU_TARGET     = at90s8535
#MCU_TARGET     = atmega128
#MCU_TARGET     = atmega1280
#MCU_TARGET     = atmega1281
#MCU_TARGET     = atmega1284p
#MCU_TARGET     = atmega16
#MCU_TARGET     = atmega163
#MCU_TARGET     = atmega164p
#MCU_TARGET     = atmega165
#MCU_TARGET     = atmega165p
#MCU_TARGET     = atmega168
#MCU_TARGET     = atmega169
#MCU_TARGET     = atmega169p
#MCU_TARGET     = atmega2560
#MCU_TARGET     = atmega2561
#MCU_TARGET     = atmega32
#MCU_TARGET     = atmega324p
#MCU_TARGET     = atmega325
#MCU_TARGET     = atmega3250
#MCU_TARGET     = atmega329
#MCU_TARGET     = atmega3290
#MCU_TARGET     = atmega32u4
#MCU_TARGET     = atmega48
#MCU_TARGET     = atmega64
#MCU_TARGET     = atmega640
#MCU_TARGET     = atmega644
#MCU_TARGET     = atmega644p
#MCU_TARGET     = atmega645
#MCU_TARGET     = atmega6450
#MCU_TARGET     = atmega649
#MCU_TARGET     = atmega6490
MCU_TARGET     = atmega8
#MCU_TARGET     = atmega8515
#MCU_TARGET     = atmega8535
#MCU_TARGET     = atmega88
#MCU_TARGET     = attiny2313
#MCU_TARGET     = attiny24
#MCU_TARGET     = attiny25
#MCU_TARGET     = attiny26
#MCU_TARGET     = attiny261
#MCU_TARGET     = attiny44
#MCU_TARGET     = attiny45
#MCU_TARGET     = attiny461
#MCU_TARGET     = attiny84
#MCU_TARGET     = attiny85
#MCU_TARGET     = attiny861
OPTIMIZE       = -O2

DEFS           =
LIBS           =

# You should not have to change anything below here.

CC             = avr-gcc

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

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

OBJCOPY        = avr-objcopy
OBJDUMP        = avr-objdump

all: $(PRG).elf lst text eeprom

$(PRG).elf: $(OBJ)
```

```
    $(CC) $(CFLAGS) $(LDFLAGS) -o $@ $^ $(LIBS)

# dependency:
demo.o: demo.c iocompat.h

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

lst:  $(PRG).lst

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

# Rules for building the .text rom images

text: hex bin srec

hex:  $(PRG).hex
bin:  $(PRG).bin
srec: $(PRG).srec

%.hex: %.elf
    $(OBJCOPY) -j .text -j .data -O ihex $< $@

%.srec: %.elf
    $(OBJCOPY) -j .text -j .data -O srec $< $@

%.bin: %.elf
    $(OBJCOPY) -j .text -j .data -O binary $< $@

# Rules for building the .eeprom rom images

eeprom: ehex ebin esrec

ehex:  $(PRG)_eeprom.hex
ebin:  $(PRG)_eeprom.bin
esrec: $(PRG)_eeprom.srec

%_eeprom.hex: %.elf
    $(OBJCOPY) -j .eeprom --change-section-lma .eeprom=0 -O ihex $< $@ \
    || { echo empty $@ not generated; exit 0; }

%_eeprom.srec: %.elf
    $(OBJCOPY) -j .eeprom --change-section-lma .eeprom=0 -O srec $< $@ \
    || { echo empty $@ not generated; exit 0; }

%_eeprom.bin: %.elf
    $(OBJCOPY) -j .eeprom --change-section-lma .eeprom=0 -O binary $< $@ \
    || { echo empty $@ not generated; exit 0; }

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

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

dox: eps png pdf

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

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

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

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

21.42.8 Reference to the source code

The source code is installed under

`$prefix/share/doc/avr-libc/examples/demo/`,

where `$prefix` is a configuration option. For Unix systems, it is usually set to either `/usr` or `/usr/local`.

21.43 A more sophisticated project

This project extends the basic idea of the simple project to control a LED with a PWM output, but adds methods to adjust the LED brightness. It employs a lot of the basic concepts of AVR-LibC to achieve that goal.

Understanding this project assumes the simple project has been understood in full, as well as being acquainted with the basic hardware concepts of an AVR microcontroller.

21.43.1 Hardware setup

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

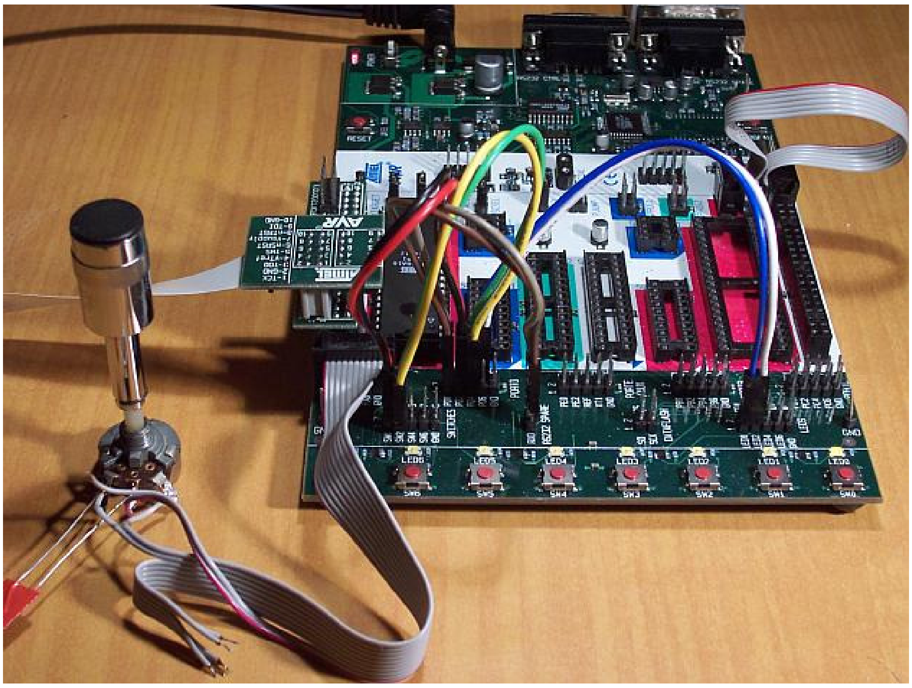

Figure 5 Setup of the STK500

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

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

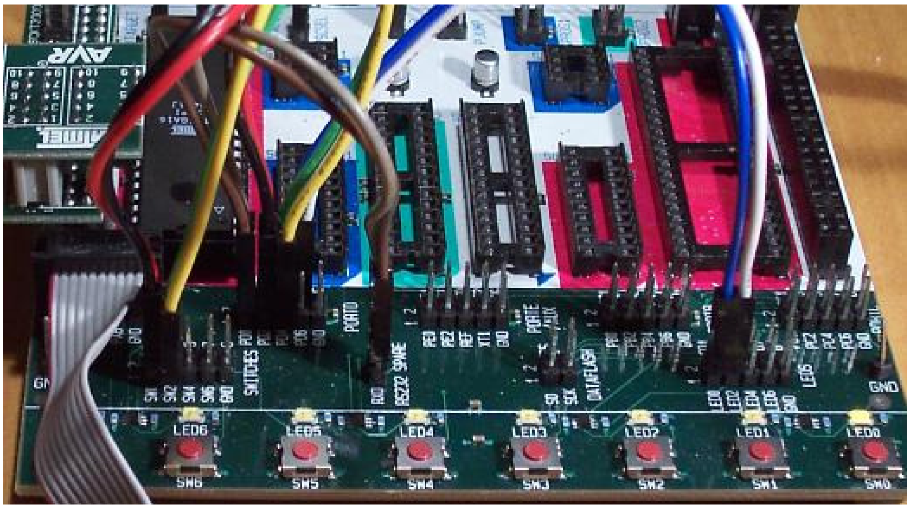

Figure 6 Wiring of the STK500

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

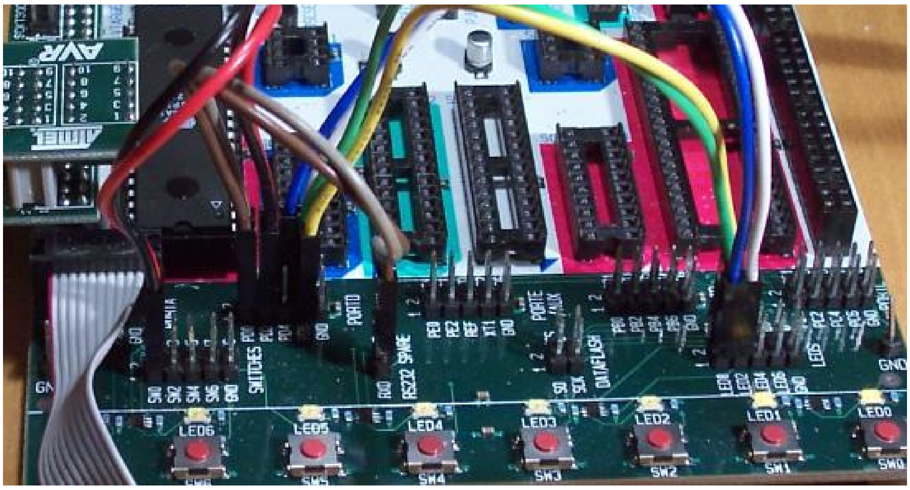

Figure 7 Wiring option #2 of the STK500

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

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

Finally, the demo has been ported to the ATtiny2313 as well. As this AVR does not offer an ADC, everything related to handling the ADC is disabled in the code for that MCU type. Also, port D of this controller type only features 6 pins, so the 1-second flash LED had to be moved from PD6 to PD4. (PD4 is used as the ADC control button on the other MCU types, but that is not needed here.) OC1A is located at PB3 on this device.

The `MCU_TARGET` macro in the Makefile needs to be adjusted appropriately for the alternative controller types.

The flash ROM and RAM consumption of this demo are way below the resources of even an ATmega48, and still well within the capabilities of an ATtiny2313. The major advantage of experimenting with the ATmega16 (in addition that it ships together with an STK500 anyway) is that it can be debugged online via JTAG. Likewise, the ATmega48/88/168 and ATtiny2313 devices can be debugged through debugWire, using the Atmel JTAG ICE mkII or the low-cost AVR Dragon.

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

21.43.2 Functional overview

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

PD0 and PD1 are configured as UART IO, and can be used to connect the demo kit to a PC (9600 Bd, 8N1 frame format). The demo application talks to the serial port, and it can be controlled from the serial port.

PD2 through PD4 are configured as inputs, and control the application unless control has been taken over by the serial port. Shorting PD2 to GND will decrease the current PWM value, shorting PD3 to GND will increase it.

While PD4 is shorted to GND, one ADC conversion for channel 0 (ADC input is on PA0) will be triggered each internal clock tick, and the resulting value will be used as the PWM value. So the brightness of the LED follows the analog input value on PC0. VAREF on the STK500 should be set to the same value as VCC.

When running in serial control mode, the function of the watchdog timer can be demonstrated by typing an `r'. This will make the demo application run in a tight loop without retriggering the watchdog so after some seconds, the watchdog will reset the MCU. This situation can be figured out on startup by reading the MCUCSR register.

The current value of the PWM is backed up in an EEPROM cell after about 3 seconds of idle time after the last change. If that EEPROM cell contains a reasonable (i. e. non-erased) value at startup, it is taken as the initial value for the PWM. This virtually preserves the last value across power cycles. By not updating the EEPROM immmediately but only after a timeout, EEPROM wear is reduced considerably compared to immediately writing the value at each change.

21.43.3 A code walkthrough

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

21.43.3.1 Part 1: Macro definitions A number of preprocessor macros are defined to improve readability and/or portability of the application.

The first macros describe the IO pins our LEDs and pushbuttons are connected to. This provides some kind of mini-HAL (hardware abstraction layer) so should some of the connections be changed, they don't need to be changed inside the code but only on top. Note that the location of the PWM output itself is mandated by the hardware, so it cannot be easily changed. As the ATmega48/88/168 controllers belong to a more recent generation of AVRs, a number of register and bit names have been changed there, so they are mapped back to their ATmega8/16 equivalents to keep the actual program code portable.

The name `F_CPU` is the conventional name to describe the CPU clock frequency of the controller. This demo project just uses the internal calibrated 1 MHz RC oscillator that is enabled by default. Note that when using the `<util/delay.h>` functions, `F_CPU` needs to be defined before including that file.

The remaining macros have their own comments in the source code. The macro `TMR1_SCALE` shows how to use the preprocessor and the compiler's constant expression computation to calculate the value of timer 1's post-scaler in a way so it only depends on `F_CPU` and the desired software clock frequency. While the formula looks a bit complicated, using a macro offers the advantage that the application will automatically scale to new target softclock or master CPU frequencies without having to manually re-calculate hardcoded constants.

21.43.3.2 Part 2: Variable definitions The `intflags` structure demonstrates a way to allocate bit variables in memory. Each of the interrupt service routines just sets one bit within that structure, and the application's main loop then monitors the bits in order to act appropriately.

Like all variables that are used to communicate values between an interrupt service routine and the main application, it is declared volatile.

The variable `ee_pwm` is not a variable in the classical C sense that could be used as an lvalue or within an expression to obtain its value. Instead, the

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

marks it as belonging to the EEPROM section. This section is merely used as a placeholder so the compiler can arrange for each individual variable's location in EEPROM. The compiler will also keep track of initial values assigned, and usually the Makefile is arranged to extract these initial values into a separate load file (`largedemo_eeprom.*` in this case) that can be used to initialize the EEPROM.

The actual EEPROM IO must be performed manually.

Similarly, the variable `mcucsr` is kept in the .noinit section in order to prevent it from being cleared upon application startup.

21.43.3.3 Part 3: Interrupt service routines The ISR to handle timer 1's overflow interrupt arranges for the software clock. While timer 1 runs the PWM, it calls its overflow handler rather frequently, so the `TMR1_SCALE` value is used as a postscaler to reduce the internal software clock frequency further. If the software clock triggers, it sets the `tmr_int` bitfield, and defers all further tasks to the main loop.

The ADC ISR just fetches the value from the ADC conversion, disables the ADC interrupt again, and announces the presence of the new value in the `adc_int` bitfield. The interrupt is kept disabled while not needed, because the ADC will also be triggered by executing the SLEEP instruction in idle mode (which is the default sleep mode). Another option would be to turn off the ADC completely here, but that increases the ADC's startup time (not that it would matter much for this application).

21.43.3.4 Part 4: Auxiliary functions The function `handle_mcucsr()` uses two `__attribute__` declarators to achieve specific goals. First, it will instruct the compiler to place the generated code into the .init3 section of the output. Thus, it will become part of the application initialization sequence. This is done in order to fetch (and clear) the reason of the last hardware reset from `MCUCSR` as early as possible. There is a short period of time where the next reset could already trigger before the current reason has been evaluated. This also explains why the variable `mcucsr` that mirrors the register's value needs to be placed into the .noinit section, because otherwise the default initialization (which happens after .init3) would blank the value again.

As the initialization code is not called using CALL/RET instructions but rather concatenated together, the compiler needs to be instructed to omit the entire function prologue and epilogue. This is performed by the *naked* attribute. So while syntactically, `handle_mcucsr()` is a function to the compiler, the compiler will just emit the instructions for it without setting up any stack frame, and not even a RET instruction at the end.

Function `ioinit()` centralizes all hardware setup. The very last part of that function demonstrates the use of the EEPROM variable `ee_pwm` to obtain an EEPROM address that can in turn be applied as an argument to `eeprom_read_word()`.

The following functions handle UART character and string output. (UART input is handled by an ISR.) There are two string output functions, `printstr()` and `printstr_p()`. The latter function fetches the string from program memory. Both functions translate a newline character into a carriage return/newline sequence, so a simple `\n` can be used in the source code.

The function `set_pwm()` propagates the new PWM value to the PWM, performing range checking. When the value has been changed, the new percentage will be announced on the serial link. The current value is mirrored in the variable `pwm` so others can use it in calculations. In order to allow for a simple calculation of a percentage value without requiring floating-point mathematics, the maximal value of the PWM is restricted to 1000 rather than 1023, so a simple division by 10 can be used. Due to the nature of the human eye, the difference in LED brightness between 1000 and 1023 is not noticable anyway.

21.43.3.5 Part 5: main()

At the start of `main()`, a variable `mode` is declared to keep the current mode of operation. An enumeration is used to improve the readability. By default, the compiler would allocate a variable of type *int* for an enumeration. The *packed* attribute declarator instructs the compiler to use the smallest possible integer type (which would be an 8-bit type here).

After some initialization actions, the application's main loop follows. In an embedded application, this is normally an infinite loop as there is nothing an application could "exit" into anyway.

At the beginning of the loop, the watchdog timer will be retriggered. If that timer is not triggered for about 2 seconds, it will issue a hardware reset. Care needs to be taken that no code path blocks longer than this, or it needs to frequently perform watchdog resets of its own. An example of such a code path would be the string IO functions: for an overly large string to print (about 2000 characters at 9600 Bd), they might block for too long.

The loop itself then acts on the interrupt indication bitfields as appropriate, and will eventually put the CPU on sleep at its end to conserve power.

The first interrupt bit that is handled is the (software) timer, at a frequency of approximately 100 Hz. The `CLOCKOUT` pin will be toggled here, so e. g. an oscilloscope can be used on that pin to measure the accuracy of our software clock. Then, the LED flasher for LED2 ("We are alive"-LED) is built. It will flash that LED for about 50 ms, and pause it for another 950 ms. Various actions depending on the operation mode follow. Finally, the 3-second backup timer is implemented that will write the PWM value back to EEPROM once it is not changing anymore.

The ADC interrupt will just adjust the PWM value only.

Finally, the UART Rx interrupt will dispatch on the last character received from the UART.

All the string literals that are used as informational messages within `main()` are placed in program memory so no SRAM needs to be allocated for them. This is done by using the PSTR macro, and passing the string to `printstr_p()`.

21.43.4 The source code

The source code is installed under

`$prefix/share/doc/avr-libc/examples/largedemo/largedemo.c`,

where `$prefix` is a configuration option. For Unix systems, it is usually set to either `/usr` or `/usr/local`.

21.44 Using the standard IO facilities

This project illustrates how to use the standard IO facilities (stdio) provided by this library. It assumes a basic knowledge of how the stdio subsystem is used in standard C applications, and concentrates on the differences in this library's implementation that mainly result from the differences of the microcontroller environment, compared to a hosted environment of a standard computer.

This demo is meant to supplement the documentation, not to replace it.

21.44.1 Hardware setup

The demo is set up in a way so it can be run on the ATmega16 that ships with the STK500 development kit. The UART port needs to be connected to the RS-232 "spare" port by a jumper cable that connects PD0 to RxD and PD1 to TxD. The RS-232 channel is set up as standard input (`stdin`) and standard output (`stdout`), respectively.

In order to have a different device available for a standard error channel (`stderr`), an industry-standard LCD display with an HD44780-compatible LCD controller has been chosen. This display needs to be connected to port A of the STK500 in the following way:

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

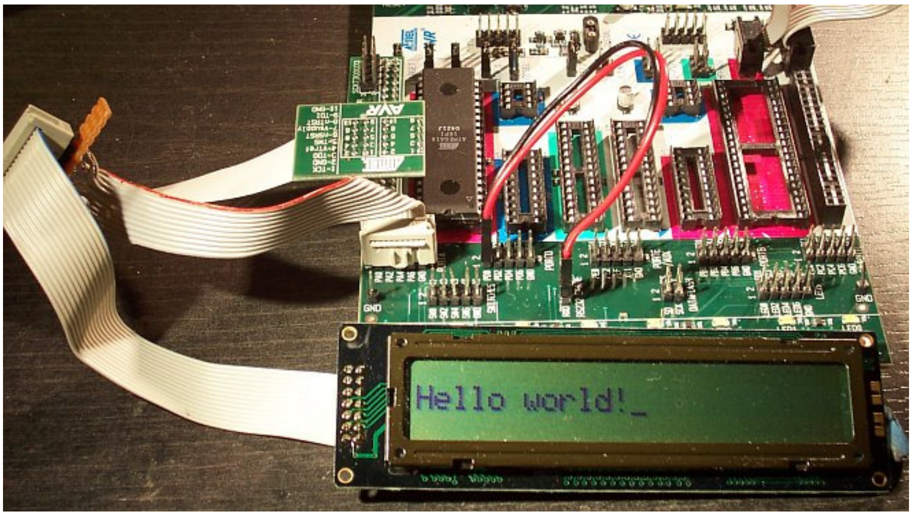

Figure 8 Wiring of the STK500

The LCD controller is used in 4-bit mode, including polling the "busy" flag so the R/∼W line from the LCD controller needs to be connected. Note that the LCD controller has yet another supply pin that is used to adjust the LCD's contrast (V5). Typically, that pin connects to a potentiometer between Vcc and GND. Often, it might work to just connect that pin to GND, while leaving it unconnected usually yields an unreadable display.

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

21.44.2 Functional overview

The project consists of the following files:

- `stdiodemo.c` This is the main example file.
- `defines.h` Contains some global defines, like the LCD wiring
- `hd44780.c` Implementation of an HD44780 LCD display driver
- `hd44780.h` Interface declarations for the HD44780 driver
- `lcd.c` Implementation of LCD character IO on top of the HD44780 driver
- `lcd.h` Interface declarations for the LCD driver
- `uart.c` Implementation of a character IO driver for the internal UART
- `uart.h` Interface declarations for the UART driver

21.44.3 A code walkthrough

21.44.3.1 stdiodemo.c As usual, include files go first. While conventionally, system header files (those in angular brackets < ... >) go before application-specific header files (in double quotes), `defines.h` comes as the first header file here. The main reason is that this file defines the value of `F_CPU` which needs to be known before including `<utils/delay.h>`.

The function `ioinit()` summarizes all hardware initialization tasks. As this function is declared to be module-internal only (`static`), the compiler will notice its simplicity, and with a reasonable optimization level in effect, it will inline that function. That needs to be kept in mind when debugging, because the inlining might cause the debugger to "jump around wildly" at a first glance when single-stepping.

The definitions of `uart_str` and `lcd_str` set up two stdio streams. The initialization is done using the `FDEV_SETUP_STREAM()` initializer template macro, so a static object can be constructed that can be used for IO purposes. This initializer macro takes three arguments, two function macros to connect the corresponding output and input functions, respectively, the third one describes the intent of the stream (read, write, or both). Those functions that are not required by the specified intent (like the input function for `lcd_str` which is specified to only perform output operations) can be given as `NULL`.

The stream `uart_str` corresponds to input and output operations performed over the RS-232 connection to a terminal (e.g. from/to a PC running a terminal program), while the `lcd_str` stream provides a method to display character data on the LCD text display.

The function `delay_1s()` suspends program execution for approximately one second. This is done using the `_delay_ms()` function from `<util/delay.h>` which in turn needs the `F_CPU` macro in order to adjust the cycle counts. As the `_delay_ms()` function has a limited range of allowable argument values (depending on `F_CPU`), a value of 10 ms has been chosen as the base delay which would be safe for CPU frequencies of up to about 26 MHz. This function is then called 100 times to accomodate for the actual one-second delay.

In a practical application, long delays like this one were better be handled by a hardware timer, so the main CPU would be free for other tasks while waiting, or could be put on sleep.

At the beginning of `main()`, after initializing the peripheral devices, the default stdio streams `stdin`, `stdout`, and `stderr` are set up by using the existing static `FILE` stream objects. While this is not mandatory, the availability of `stdin` and `stdout` allows to use the shorthand functions (e.g. `printf()` instead of `fprintf()`), and `stderr` can mnemonically be referred to when sending out diagnostic messages.

Just for demonstration purposes, `stdin` and `stdout` are connected to a stream that will perform UART IO, while `stderr` is arranged to output its data to the LCD text display.

Finally, a main loop follows that accepts simple "commands" entered via the RS-232 connection, and performs a few simple actions based on the commands.

First, a prompt is sent out using `printf_P()` (which takes a program space string). The string is read into an internal buffer as one line of input, using `fgets()`. While it would be also possible to use `gets()` (which implicitly reads from `stdin`), `gets()` has no control that the user's input does not overflow the input buffer provided so it should never be used at all.

If `fgets()` fails to read anything, the main loop is left. Of course, normally the main loop of a microcontroller application is supposed to never finish, but again, for demonstrational purposes, this explains the error handling of stdio. `fgets()` will return NULL in case of an input error or end-of-file condition on input. Both these conditions are in the domain of the function that is used to establish the stream, `uart_putchar()` in this case. In short, this function returns EOF in case of a serial line "break" condition (extended start condition) has been recognized on the serial line. Common PC terminal programs allow to assert this condition as some kind of out-of-band signalling on an RS-232 connection.

When leaving the main loop, a goodbye message is sent to standard error output (i.e. to the LCD), followed by three dots in one-second spacing, followed by a sequence that will clear the LCD. Finally, `main()` will be terminated, and the library will add an infinite loop, so only a CPU reset will be able to restart the application.

There are three "commands" recognized, each determined by the first letter of the line entered (converted to lower case):

- The 'q' (quit) command has the same effect of leaving the main loop.
- The 'l' (LCD) command takes its second argument, and sends it to the LCD.
- The 'u' (UART) command takes its second argument, and sends it back to the UART connection.

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

21.44.3.2 defines.h

This file just contains a few peripheral definitions.

The `F_CPU` macro defines the CPU clock frequency, to be used in delay loops, as well as in the UART baud rate calculation.

The macro `UART_BAUD` defines the RS-232 baud rate. Depending on the actual CPU frequency, only a limited range of baud rates can be supported.

The remaining macros customize the IO port and pins used for the HD44780 LCD driver. Each definition consists of a letter naming the port this pin is attached to, and a respective bit number. For accessing the data lines, only the first data line gets its own macro (line D4 on the HD44780, lines D0 through D3 are not used in 4-bit mode), all other data lines are expected to be in ascending order next to D4.

21.44.3.3 hd44780.h

This file describes the public interface of the low-level LCD driver that interfaces to the HD44780 LCD controller. Public functions are available to initialize the controller into 4-bit mode, to wait for the controller's busy bit to be clear, and to read or write one byte from or to the controller.

As there are two different forms of controller IO, one to send a command or receive the controller status (RS signal clear), and one to send or receive data to/from the controller's SRAM (RS asserted), macros are provided that build on the mentioned function primitives.

Finally, macros are provided for all the controller commands to allow them to be used symbolically. The HD44780 datasheet explains these basic functions of the controller in more detail.

21.44.3.4 hd44780.c

This is the implementation of the low-level HD44780 LCD controller driver.

On top, a few preprocessor glueing tricks are used to establish symbolic access to the hardware port pins the LCD controller is attached to, based on the application's definitions made in defines.h.

The `hd44780_pulse_e()` function asserts a short pulse to the controller's E (enable) pin. Since reading back the data asserted by the LCD controller needs to be performed while E is active, this function reads and returns the input data if the parameter `readback` is true. When called with a compile-time constant parameter that is false, the compiler will completely eliminate the unused readback operation, as well as the return value as part of its optimizations.

As the controller is used in 4-bit interface mode, all byte IO to/from the controller needs to be handled as two nibble IOs. The functions `hd44780_outnibble()` and `hd44780_innibble()` implement this. They do not belong to the public interface, so they are declared static.

Building upon these, the public functions `hd44780_outbyte()` and `hd44780_inbyte()` transfer one byte to/from the controller.

The function `hd44780_wait_ready()` waits for the controller to become ready, by continuously polling the controller's status (which is read by performing a byte read with the RS signal cleard), and examining the BUSY flag within the status byte. This function needs to be called before performing any controller IO.

Finally, `hd44780_init()` initializes the LCD controller into 4-bit mode, based on the initialization sequence mandated by the datasheet. As the BUSY flag cannot be examined yet at this point, this is the only part of this code where timed delays are used. While the controller can perform a power-on reset when certain constraints on the power supply rise time are met, always calling the software initialization routine at startup ensures the controller will be in a known state. This function also puts the interface into 4-bit mode (which would not be done automatically after a power-on reset).

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

21.44.3.6 lcd.c The implementation of the higher-level LCD driver. This driver builds on top of the HD44780 low-level LCD controller driver, and offers a character IO interface suitable for direct use by the standard IO facilities. Where the low-level HD44780 driver deals with setting up controller SRAM addresses, writing data to the controller's SRAM, and controlling display functions like clearing the display, or moving the cursor, this high-level driver allows to just write a character to the LCD, in the assumption this will somehow show up on the display.

Control characters can be handled at this level, and used to perform specific actions on the LCD. Currently, there is only one control character that is being dealt with: a newline character (`\n`) is taken as an indication to clear the display and set the cursor into its initial position upon reception of the next character, so a "new line" of text can be displayed. Therefore, a received newline character is remembered until more characters have been sent by the application, and will only then cause the display to be cleared before continuing. This provides a convenient abstraction where full lines of text can be sent to the driver, and will remain visible at the LCD until the next line is to be displayed.

Further control characters could be implemented, e. g. using a set of escape sequences. That way, it would be possible to implement self-scrolling display lines etc.

The public function `lcd_init()` first calls the initialization entry point of the lower-level HD44780 driver, and then sets up the LCD in a way we'd like to (display cleared, non-blinking cursor enabled, SRAM addresses are increasing so characters will be written left to right).

The public function `lcd_putchar()` takes arguments that make it suitable for being passed as a `put()` function pointer to the stdio stream initialization functions and macros (`fdevopen()`, `FDEV_SETUP_STREAM()` etc.). Thus, it takes two arguments, the character to display itself, and a reference to the underlying stream object, and it is expected to return 0 upon success.

This function remembers the last unprocessed newline character seen in the function-local static variable `nl_seen`. If a newline character is encountered, it will simply set this variable to a true value, and return to the caller. As soon as the first non-newline character is to be displayed with `nl_seen` still true, the LCD controller is told to clear the display, put the cursor home, and restart at SRAM address 0. All other characters are sent to the display.

The single static function-internal variable `nl_seen` works for this purpose. If multiple LCDs should be controlled using the same set of driver functions, that would not work anymore, as a way is needed to distinguish between the various displays. This is where the second parameter can be used, the reference to the stream itself: instead of keeping the state inside a private variable of the function, it can be kept inside a private object that is attached to the stream itself. A reference to that private object can be attached to the stream (e.g. inside the function `lcd_init()` that then also needs to be passed a reference to the stream) using `fdev_set_udata()`, and can be accessed inside `lcd_putchar()` using fdev_get_udata().

21.44.3.7 uart.h Public interface definition for the RS-232 UART driver, much like in lcd.h except there is now also a character input function available.

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

21.44.3.8 uart.c This implements an stdio-compatible RS-232 driver using an AVR's standard UART (or USART in asynchronous operation mode). Both, character output as well as character input operations are implemented. Character output takes care of converting the internal newline `\n` into its external representation carriage return/line feed (`\r\n`).

Character input is organized as a line-buffered operation that allows to minimally edit the current line until it is "sent" to the application when either a carriage return (`\r`) or newline (`\n`) character is received from the terminal. The line editing functions implemented are:

- \b (back space) or \177 (delete) deletes the previous character
- ^u (control-U, ASCII NAK) deletes the entire input buffer
- ^w (control-W, ASCII ETB) deletes the previous input word, delimited by white space
- ^r (control-R, ASCII DC2) sends a \r, then reprints the buffer (refresh)
- \t (tabulator) will be replaced by a single space

The function `uart_init()` takes care of all hardware initialization that is required to put the UART into a mode with 8 data bits, no parity, one stop bit (commonly referred to as 8N1) at the baud rate configured in defines.h. At low CPU clock frequencies, the `U2X` bit in the UART is set, reducing the oversampling from 16x to 8x, which allows for a 9600 Bd rate to be achieved with tolerable error using the default 1 MHz RC oscillator.

The public function `uart_putchar()` again has suitable arguments for direct use by the stdio stream interface. It performs the \n into \r\n translation by recursively calling itself when it sees a \n character. Just for demonstration purposes, the \a (audible bell, ASCII BEL) character is implemented by sending a string to `stderr`, so it will be displayed on the LCD.

The public function `uart_getchar()` implements the line editor. If there are characters available in the line buffer (variable `rxp` is not `NULL`), the next character will be returned from the buffer without any UART interaction.

If there are no characters inside the line buffer, the input loop will be entered. Characters will be read from the UART, and processed accordingly. If the UART signalled a framing error (`FE` bit set), typically caused by the terminal sending a *line break* condition (start condition held much longer than one character period), the function will return an end-of-file condition using `_FDEV_EOF`. If there was a data overrun condition on input (`DOR` bit set), an error condition will be returned as `_FDEV_ERR`.

Line editing characters are handled inside the loop, potentially modifying the buffer status. If characters are attempted to be entered beyond the size of the line buffer, their reception is refused, and a \a character is sent to the terminal. If a \r or \n character is seen, the variable `rxp` (receive pointer) is set to the beginning of the buffer, the loop is left, and the first character of the buffer will be returned to the application. (If no other characters have been entered, this will just be the newline character, and the buffer is marked as being exhausted immediately again.)

21.44.4 The source code

The source code is installed under

`$prefix/share/doc/avr-libc/examples/stdiodemo/`,

where `$prefix` is a configuration option. For Unix systems, it is usually set to either `/usr` or `/usr/local`.

21.45 Example using the two-wire interface (TWI)

Some newer devices of the ATmega series contain builtin support for interfacing the microcontroller to a two-wire bus, called TWI. This is essentially the same called I2C by Philips, but that term is avoided in Atmel's documentation due to patenting issues.

For further documentation, see:

http://www.nxp.com/documents/user_manual/UM10204.pdf

21.45.1 Introduction into TWI

The two-wire interface consists of two signal lines named *SDA* (serial data) and *SCL* (serial clock) (plus a ground line, of course). All devices participating in the bus are connected together, using open-drain driver circuitry, so the wires must be terminated using appropriate pullup resistors. The pullups must be small enough to recharge the line capacity in short enough time compared to the desired maximal clock frequency, yet large enough so all drivers will not be overloaded. There are formulas in the datasheet that help selecting the pullups.

Devices can either act as a master to the bus (i. e., they initiate a transfer), or as a slave (they only act when being called by a master). The bus is multi-master capable, and a particular device implementation can act as either master or slave at different times. Devices are addressed using a 7-bit address (coordinated by Philips) transfered as the first byte after the so-called start condition. The LSB of that byte is R/∼W, i. e. it determines whether the request to the slave is to read or write data during the next cycles. (There is also an option to have devices using 10-bit addresses but that is not covered by this example.)

21.45.2 The TWI example project

The ATmega TWI hardware supports both, master and slave operation. This example will only demonstrate how to use an AVR microcontroller as TWI master. The implementation is kept simple in order to concentrate on the steps that are required to talk to a TWI slave, so all processing is done in polled-mode, waiting for the TWI interface to indicate that the next processing step is due (by setting the TWINT interrupt bit). If it is desired to have the entire TWI communication happen in "background", all this can be implemented in an interrupt-controlled way, where only the start condition needs to be triggered from outside the interrupt routine.

There is a variety of slave devices available that can be connected to a TWI bus. For the purpose of this example, an EEPROM device out of the industry-standard **24C*xx*** series has been chosen (where *xx* can be one of **01**, **02**, **04**, **08**, or **16**) which are available from various vendors. The choice was almost arbitrary, mainly triggered by the fact that an EEPROM device is being talked to in both directions, reading and writing the slave device, so the example will demonstrate the details of both.

Usually, there is probably not much need to add more EEPROM to an ATmega system that way: the smallest possible AVR device that offers hardware TWI support is the ATmega8 which comes with 512 bytes of EEPROM, which is equivalent to an 24C04 device. The ATmega128 already comes with twice as much EEPROM as the 24C16 would offer. One exception might be to use an externally connected EEPROM device that is removable; e. g. SDRAM PC memory comes with an integrated TWI EEPROM that carries the RAM configuration information.

21.45.3 The Source Code

The source code is installed under

`$prefix/share/doc/avr-libc/examples/twitest/twitest.c`,

where `$prefix` is a configuration option. For Unix systems, it is usually set to either `/usr` or `/usr/local`.

Note [1]

The header file `<util/twi.h>` contains some macro definitions for symbolic constants used in the TWI status register. These definitions match the names used in the Atmel datasheet except that all names have been prefixed with `TW_`.

Note [2]

The clock is used in timer calculations done by the compiler, for the UART baud rate and the TWI clock rate.

Note [3]

The address assigned for the 24Cxx EEPROM consists of 1010 in the upper four bits. The following three bits are normally available as slave sub-addresses, allowing to operate more than one device of the same type on a single bus, where the actual subaddress used for each device is configured by hardware strapping. However, since the next data packet following the device selection only allows for 8 bits that are used as an EEPROM address, devices that require more than 8 address bits (24C04 and above) "steal" subaddress bits and use them for the EEPROM cell address bits 9 to 11 as required. This example simply assumes all subaddress bits are 0 for the smaller devices, so the E0, E1, and E2 inputs of the 24Cxx must be grounded.

Note [3a]

EEPROMs of type 24C32 and above cannot be addressed anymore even with the subaddress bit trick. Thus, they require the upper address bits being sent separately on the bus. When activating the `WORD_ADDRESS_16BIT` define, the algorithm implements that auxiliary address byte transmission.

Note [4]

For slow clocks, enable the 2 x U[S]ART clock multiplier, to improve the baud rate error. This will allow a 9600 Bd communication using the standard 1 MHz calibrated RC oscillator. See also the Baud rate tables in the datasheets.

Note [5]

The datasheet explains why a minimum TWBR value of 10 should be maintained when running in master mode. Thus, for system clocks below 3.6 MHz, we cannot run the bus at the intented clock rate of 100 kHz but have to slow down accordingly.

Note [6]

This function is used by the standard output facilities that are utilized in this example for debugging and demonstration purposes.

Note [7]

In order to shorten the data to be sent over the TWI bus, the 24Cxx EEPROMs support multiple data bytes transfered within a single request, maintaining an internal address counter that is updated after each data byte transfered successfully. When reading data, one request can read the entire device memory if desired (the counter would wrap around and start back from 0 when reaching the end of the device).

Note [8]

When reading the EEPROM, a first device selection must be made with write intent (R/∼W bit set to 0 indicating a write operation) in order to transfer the EEPROM address to start reading from. This is called *master transmitter mode*. Each completion of a particular step in TWI communication is indicated by an asserted TWINT bit in TWCR. (An interrupt would be generated if allowed.) After performing any actions that are needed for the next communication step, the interrupt condition must be manually cleared by *setting* the TWINT bit. Unlike with many other interrupt sources, this would even be required when using a true interrupt routine, since as soon as TWINT is re-asserted, the next bus transaction will start.

Note [9]

Since the TWI bus is multi-master capable, there is potential for a bus contention when one master starts to access the bus. Normally, the TWI bus interface unit will detect this situation, and will not initiate a start condition while the bus is busy. However, in case two masters were starting at exactly the same time, the way bus arbitration works, there is always a chance that one master could lose arbitration of the bus during any transmit operation. A master that has lost arbitration is required by the protocol to immediately cease talking on the bus; in particular it must not initiate a stop condition in order to not corrupt the ongoing transfer from the active master. In this example, upon detecting a lost arbitration condition, the entire transfer is going to be restarted. This will cause a new start condition to be initiated, which will normally be delayed until the currently active master has released the bus.

Note [10]

Next, the device slave is going to be reselected (using a so-called repeated start condition which is meant to guarantee that the bus arbitration will remain at the current master) using the same slave address (SLA), but this time with read intent (R/∼W bit set to 1) in order to request the device slave to start transfering data from the slave to the master in the next packet.

Note [11]

If the EEPROM device is still busy writing one or more cells after a previous write request, it will simply leave its bus interface drivers at high impedance, and does not respond to a selection in any way at all. The master selecting the device will see the high level at SDA after transfering the SLA+R/W packet as a NACK to its selection request. Thus, the select process is simply started over (effectively causing a *repeated start condition*), until the device will eventually respond. This polling procedure is recommended in the 24Cxx datasheet in order to minimize the busy wait time when writing. Note that in case a device is broken and never responds to a selection (e. g. since it is no longer present at all), this will cause an infinite loop. Thus the maximal number of iterations made until the device is declared to be not responding at all, and an error is returned, will be limited to MAX_ITER.

Note [12]

This is called *master receiver mode*: the bus master still supplies the SCL clock, but the device slave drives the SDA line with the appropriate data. After 8 data bits, the master responds with an ACK bit (SDA driven low) in order to request another data transfer from the slave, or it can leave the SDA line high (NACK), indicating to the slave that it is going to stop the transfer now. Assertion of ACK is handled by setting the TWEA bit in TWCR when starting the current transfer.

Note [13]

The control word sent out in order to initiate the transfer of the next data packet is initially set up to assert the TWEA bit. During the last loop iteration, TWEA is de-asserted so the client will get informed that no further transfer is desired.

Note [14]

Except in the case of lost arbitration, all bus transactions must properly be terminated by the master initiating a stop condition.

Note [15]

Writing to the EEPROM device is simpler than reading, since only a master transmitter mode transfer is needed. Note that the first packet after the SLA+W selection is always considered to be the EEPROM address for the next operation. (This packet is exactly the same as the one above sent before starting to read the device.) In case a master transmitter mode transfer is going to send more than one data packet, all following packets will be considered data bytes to write at the indicated address. The internal address pointer will be incremented after each write operation.

Note [16]

24Cxx devices can become write-protected by strapping their ~WC pin to logic high. (Leaving it unconnected is explicitly allowed, and constitutes logic low level, i. e. no write protection.) In case of a write protected device, all data transfer attempts will be NACKed by the device. Note that some devices might not implement this.

22 Data Structure Documentation

22.1 div_t Struct Reference

```
#include <stdlib.h>
```

Data Fields

- int quot
- int rem

22.1.1 Detailed Description

Result type for function div().

22.1.2 Field Documentation

22.1.2.1 quot `int div_t::quot`

The Quotient.

22.1.2.2 rem `int div_t::rem`

The Remainder.

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

- stdlib.h

22.2 ldiv_t Struct Reference

```
#include <stdlib.h>
```

Data Fields

- long quot
- long rem

22.2.1 Detailed Description

Result type for function ldiv().

22.2.2 Field Documentation

22.2.2.1 quot `long ldiv_t::quot`

The Quotient.

22.2.2.2 rem `long ldiv_t::rem`

The Remainder.

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

- stdlib.h

22.3 tm Struct Reference

```
#include <time.h>
```

Data Fields

- int8_t tm_sec
- int8_t tm_min
- int8_t tm_hour
- int8_t tm_mday
- int8_t tm_wday
- int8_t tm_mon
- int16_t tm_year
- int16_t tm_yday
- int16_t tm_isdst

22.3.1 Detailed Description

The tm structure contains a representation of time 'broken down' into components of the Gregorian calendar.

The value of tm_isdst is zero if Daylight Saving Time is not in effect, and is negative if the information is not available.

When Daylight Saving Time is in effect, the value represents the number of seconds the clock is advanced.

See the set_dst() function for more information about Daylight Saving.

22.3.2 Field Documentation

22.3.2.1 tm_hour `int8_t tm::tm_hour`

hours since midnight - [0 to 23]

22.3.2.2 tm_isdst `int16_t tm::tm_isdst`

Daylight Saving Time flag

22.3.2.3 tm_mday `int8_t tm::tm_mday`

day of the month - [1 to 31]

22.3.2.4 tm_min `int8_t tm::tm_min`

minutes after the hour - [0 to 59]

22.3.2.5 tm_mon `int8_t tm::tm_mon`

months since January - [0 to 11]

22.3.2.6 tm_sec `int8_t tm::tm_sec`

seconds after the minute - [0 to 59]

22.3.2.7 tm_wday `int8_t tm::tm_wday`

days since Sunday - [0 to 6]

22.3.2.8 tm_yday `int16_t tm::tm_yday`

days since January 1 - [0 to 365]

22.3.2.9 tm_year `int16_t tm::tm_year`

years since 1900

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

- time.h

22.4 week_date Struct Reference

`#include <time.h>`

Data Fields

- int year
- int week
- int day

22.4.1 Detailed Description

Structure which represents a date as a year, week number of that year, and day of week. See `http://en.wikipedia.org/wiki/ISO_week_date` for more information.

22.4.2 Field Documentation

22.4.2.1 day `int week_date::day`

day within week

22.4.2.2 week `int week_date::week`

week number (#1 is where first Thursday is in)

22.4.2.3 year `int week_date::year`

year number (Gregorian calendar)

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

- time.h

23 File Documentation

23.1 project.h

```
/*
 * ----------------------------------------------------------------------------
 * "THE BEER-WARE LICENSE" (Revision 42):
 * Joerg Wunsch 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
 * ----------------------------------------------------------------------------
 *
 * Demo combining C and assembly source files.
 *
 * $Id$
 */

/*
 * Global register variables.
 */
#ifdef __ASSEMBLER__

#  define sreg_save r2
#  define flags     r16
#  define counter_hi    r4

#else  /* !ASSEMBLER */

#include <stdint.h>

register uint8_t sreg_save asm("r2");
register uint8_t flags     asm("r16");
register uint8_t counter_hi asm("r4");

#endif /* ASSEMBLER */
```

23.2 iocompat.h

```
/*
 * ----------------------------------------------------------------------------
 * "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
 * ----------------------------------------------------------------------------
 *
 * IO feature compatibility definitions for various AVRs.
 *
 * $Id$
 */
```

```
#if !defined(IOCOMPAT_H)
#define IOCOMPAT_H 1

/*
 * Device-specific adjustments:
 *
 * Supply definitions for the location of the OCR1[A] port/pin, the
 * name of the OCR register controlling the PWM, and adjust interrupt
 * vector names that differ from the one used in demo.c
 * [TIMER1_OVF_vect].
 */
#if defined(__AVR_AT90S2313__)
#  define OC1 PB3
#  define OCR OCR1
#  define DDROC DDRB
#  define TIMER1_OVF_vect TIMER1_OVF1_vect
#elif defined(__AVR_AT90S2333__) || defined(__AVR_AT90S4433__)
#  define OC1 PB1
#  define DDROC DDRB
#  define OCR OCR1
#elif defined(__AVR_AT90S4414__) || defined(__AVR_AT90S8515__) || \
      defined(__AVR_AT90S4434__) || defined(__AVR_AT90S8535__) || \
      defined(__AVR_ATmega163__) || defined(__AVR_ATmega8515__) || \
      defined(__AVR_ATmega8535__) || \
      defined(__AVR_ATmega164P__) || defined(__AVR_ATmega324P__) || \
      defined(__AVR_ATmega644__) || defined(__AVR_ATmega644P__) || \
      defined(__AVR_ATmega1284P__)
#  define OC1 PD5
#  define DDROC DDRD
#  define OCR OCR1A
#  if !defined(TIMSK)       /* new ATmegas */
#    define TIMSK TIMSK1
#  endif
#elif defined(__AVR_ATmega8__) || defined(__AVR_ATmega48__) || \
      defined(__AVR_ATmega88__) || defined(__AVR_ATmega168__)
#  define OC1 PB1
#  define DDROC DDRB
#  define OCR OCR1A
#  if !defined(TIMSK)       /* ATmega48/88/168 */
#    define TIMSK TIMSK1
#  endif /* !defined(TIMSK) */
#elif defined(__AVR_ATtiny2313__)
#  define OC1 PB3
#  define OCR OCR1A
#  define DDROC DDRB
#elif defined(__AVR_ATtiny24__) || defined(__AVR_ATtiny44__) || \
      defined(__AVR_ATtiny84__)
#  define OC1 PA6
#  define DDROC DDRA
#  if !defined(OCR1A)
#    /* work around misspelled name in AVR-LibC 1.4.[0..1] */
#    define OCR OCRA1
#  else
#    define OCR OCR1A
#  endif
#  define TIMSK TIMSK1
#  define TIMER1_OVF_vect TIM1_OVF_vect /* XML and datasheet mismatch */
#elif defined(__AVR_ATtiny25__) || defined(__AVR_ATtiny45__) || \
      defined(__AVR_ATtiny85__)
/* Timer 1 is only an 8-bit timer on these devices. */
#  define OC1 PB1
#  define DDROC DDRB
#  define OCR OCR1A
#  define TCCR1A TCCR1
#  define TCCR1B TCCR1
#  define TIMER1_OVF_vect TIM1_OVF_vect
#  define TIMER1_TOP 255    /* only 8-bit PWM possible */
#  define TIMER1_PWM_INIT _BV(PWM1A) | _BV(COM1A1)
#  define TIMER1_CLOCKSOURCE _BV(CS12) /* use 1/8 prescaler */
#elif defined(__AVR_ATtiny26__)
/* Rather close to ATtinyX5 but different enough for its own section. */
#  define OC1 PB1
#  define DDROC DDRB
#  define OCR OCR1A
#  define TIMER1_OVF_vect TIMER1_OVF1_vect
#  define TIMER1_TOP 255    /* only 8-bit PWM possible */
#  define TIMER1_PWM_INIT _BV(PWM1A) | _BV(COM1A1)
#  define TIMER1_CLOCKSOURCE _BV(CS12) /* use 1/8 prescaler */
/*
 * Without setting OCR1C to TOP, the ATtiny26 does not trigger an
 * overflow interrupt in PWM mode.
 */
#  define TIMER1_SETUP_HOOK() OCR1C = 255
#elif defined(__AVR_ATtiny261__) || defined(__AVR_ATtiny461__) || \
      defined(__AVR_ATtiny861__)
#  define OC1 PB1
#  define DDROC DDRB
```

```
#  define OCR OCR1A
#  define TIMER1_PWM_INIT _BV(WGM10) | _BV(PWM1A) | _BV(COM1A1)
/*
 * While timer 1 could be operated in 10-bit mode on these devices,
 * the handling of the 10-bit IO registers is more complicated than
 * that of the 16-bit registers of other AVR devices (no combined
 * 16-bit IO operations possible), so we restrict this demo to 8-bit
 * mode which is pretty standard.
 */
#  define TIMER1_TOP 255
#  define TIMER1_CLOCKSOURCE _BV(CS12) /* use 1/8 prescaler */
#elif defined(__AVR_ATmega32__) || defined(__AVR_ATmega16__)
#  define OC1 PD5
#  define DDROC DDRD
#  define OCR OCR1A
#elif defined(__AVR_ATmega64__) || defined(__AVR_ATmega128__) || \
      defined(__AVR_ATmega165__) || defined(__AVR_ATmega169__) || \
      defined(__AVR_ATmega325__) || defined(__AVR_ATmega3250__) || \
      defined(__AVR_ATmega645__) || defined(__AVR_ATmega6450__) || \
      defined(__AVR_ATmega329__) || defined(__AVR_ATmega3290__) || \
      defined(__AVR_ATmega649__) || defined(__AVR_ATmega6490__) || \
      defined(__AVR_ATmega640__) || \
      defined(__AVR_ATmega1280__) || defined(__AVR_ATmega1281__) || \
      defined(__AVR_ATmega2560__) || defined(__AVR_ATmega2561__) || \
      defined(__AVR_ATmega32U4__)
#  define OC1 PB5
#  define DDROC DDRB
#  define OCR OCR1A
#  if !defined(PB5)          /* work around missing bit definition */
#    define PB5 5
#  endif
#  if !defined(TIMSK)        /* new ATmegas */
#    define TIMSK TIMSK1
#  endif
#else
#  error "Don't know what kind of MCU you are compiling for"
#endif

/*
 * Map register names for older AVRs here.
 */
#if !defined(COM1A1)
#  define COM1A1 COM11
#endif

#if !defined(WGM10)
#  define WGM10 PWM10
#  define WGM11 PWM11
#endif

/*
 * Provide defaults for device-specific macros unless overridden
 * above.
 */
#if !defined(TIMER1_TOP)
#  define TIMER1_TOP 1023   /* 10-bit PWM */
#endif

#if !defined(TIMER1_PWM_INIT)
#  define TIMER1_PWM_INIT _BV(WGM10) | _BV(WGM11) | _BV(COM1A1)
#endif

#if !defined(TIMER1_CLOCKSOURCE)
#  define TIMER1_CLOCKSOURCE _BV(CS10) /* full clock */
#endif

#endif /* !defined(IOCOMPAT_H) */
```

23.3 defines.h

```
/*
 * ----------------------------------------------------------------------------
 * "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
 * ----------------------------------------------------------------------------
 *
 * General stdiodemo defines
 *
 * $Id$
 */

/* CPU frequency */
```

```
#define F_CPU 1000000UL

/* UART baud rate */
#define UART_BAUD  9600

/* HD44780 LCD port connections */
#define HD44780_RS A, 6
#define HD44780_RW A, 4
#define HD44780_E  A, 5
/* The data bits have to be not only in ascending order but also consecutive. */
#define HD44780_D4 A, 0

/* Whether to read the busy flag, or fall back to
   worst-time delays. */
#define USE_BUSY_BIT 1
```

23.4 hd44780.h

```
/*
 * ----------------------------------------------------------------------------
 * "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
 * ----------------------------------------------------------------------------
 *
 * HD44780 LCD display driver
 *
 * $Id$
 */

/*
 * Send byte b to the LCD.  rs is the RS signal (register select), 0
 * selects instruction register, 1 selects the data register.
 */
void    hd44780_outbyte(uint8_t b, uint8_t rs);

/*
 * Read one byte from the LCD controller.  rs is the RS signal, 0
 * selects busy flag (bit 7) and address counter, 1 selects the data
 * register.
 */
uint8_t  hd44780_inbyte(uint8_t rs);

/*
 * Wait for the busy flag to clear.
 */
void    hd44780_wait_ready(bool islong);

/*
 * Initialize the LCD controller hardware.
 */
void    hd44780_init(void);

/*
 * Prepare the LCD controller pins for powerdown.
 */
void    hd44780_powerdown(void);


/* Send a command to the LCD controller. */
#define hd44780_outcmd(n)   hd44780_outbyte((n), 0)

/* Send a data byte to the LCD controller. */
#define hd44780_outdata(n)  hd44780_outbyte((n), 1)

/* Read the address counter and busy flag from the LCD. */
#define hd44780_incmd()     hd44780_inbyte(0)

/* Read the current data byte from the LCD. */
#define hd44780_indata()    hd44780_inbyte(1)


/* Clear LCD display command. */
#define HD44780_CLR \
    0x01

/* Home cursor command. */
#define HD44780_HOME \
    0x02

/*
 * Select the entry mode.  inc determines whether the address counter
 * auto-increments, shift selects an automatic display shift.
```

```
 */
#define HD44780_ENTMODE(inc, shift) \
    (0x04 | ((inc)? 0x02: 0) | ((shift)? 1: 0))

/*
 * Selects disp[lay] on/off, cursor on/off, cursor blink[ing]
 * on/off.
 */
#define HD44780_DISPCTL(disp, cursor, blink) \
    (0x08 | ((disp)? 0x04: 0) | ((cursor)? 0x02: 0) | ((blink)? 1: 0))

/*
 * With shift = 1, shift display right or left.
 * With shift = 0, move cursor right or left.
 */
#define HD44780_SHIFT(shift, right) \
    (0x10 | ((shift)? 0x08: 0) | ((right)? 0x04: 0))

/*
 * Function set.  if8bit selects an 8-bit data path, twoline arranges
 * for a two-line display, font5x10 selects the 5x10 dot font (5x8
 * dots if clear).
 */
#define HD44780_FNSET(if8bit, twoline, font5x10) \
    (0x20 | ((if8bit)? 0x10: 0) | ((twoline)? 0x08: 0) | \
        ((font5x10)? 0x04: 0))

/*
 * Set the next character generator address to addr.
 */
#define HD44780_CGADDR(addr) \
    (0x40 | ((addr) & 0x3f))

/*
 * Set the next display address to addr.
 */
#define HD44780_DDADDR(addr) \
    (0x80 | ((addr) & 0x7f))

```

23.5 lcd.h

```
/*
 * ----------------------------------------------------------------------------
 * "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
 * ----------------------------------------------------------------------------
 *
 * Stdio demo, upper layer of LCD driver.
 *
 * $Id$
 */

/*
 * Initialize LCD controller.  Performs a software reset.
 */
void    lcd_init(void);

/*
 * Send one character to the LCD.
 */
int lcd_putchar(char c, FILE *stream);
```

23.6 uart.h

```
/*
 * ----------------------------------------------------------------------------
 * "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
 * ----------------------------------------------------------------------------
 *
 * Stdio demo, UART declarations
 *
 * $Id$
 */

/*
```

```
 * Perform UART startup initialization.
 */
void    uart_init(void);

/*
 * Send one character to the UART.
 */
int uart_putchar(char c, FILE *stream);

/*
 * Size of internal line buffer used by uart_getchar().
 */
#define RX_BUFSIZE 80

/*
 * Receive one character from the UART.  The actual reception is
 * line-buffered, and one character is returned from the buffer at
 * each invokation.
 */
int uart_getchar(FILE *stream);
```

23.7 alloca.h

```
/* Copyright (c) 2007, Dmitry Xmelkov
   All rights reserved.

   Redistribution and use in source and binary forms, with or without
   modification, are permitted provided that the following conditions are met:

   * Redistributions of source code must retain the above copyright
     notice, this list of conditions and the following disclaimer.
   * Redistributions in binary form must reproduce the above copyright
     notice, this list of conditions and the following disclaimer in
     the documentation and/or other materials provided with the
     distribution.
   * Neither the name of the copyright holders nor the names of
     contributors may be used to endorse or promote products derived
     from this software without specific prior written permission.

  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
  AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
  IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
  ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
  LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
  CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
  SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
  INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
  CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
  ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
  POSSIBILITY OF SUCH DAMAGE. */

/* $Id$ */

#ifndef _ALLOCA_H
#define _ALLOCA_H   1

#include <stddef.h>

/** \defgroup alloca <alloca.h>: Allocate space in the stack    */

/** \ingroup alloca
    \brief Allocate \a __size bytes of space in the stack frame of the caller.

    This temporary space is automatically freed when the function that
    called alloca() returns to its caller. AVR-LibC defines the alloca() as
    a macro, which is translated into the inlined \c __builtin_alloca()
    function. The fact that the code is inlined, means that it is impossible
    to take the address of this function, or to change its behaviour by
    linking with a different library.

    \return alloca() returns a pointer to the beginning of the allocated
    space. If the allocation causes stack overflow, program behaviour is
    undefined.

    \warning Avoid use alloca() inside the list of arguments of a function
    call.
 */
extern void *alloca (size_t __size);

#define alloca(size)    __builtin_alloca (size)

#endif  /* alloca.h */
```

23.8 assert.h File Reference

Macros

- #define assert(expression)

23.9 assert.h

Go to the documentation of this file.

```
/* Copyright (c) 2005,2007 Joerg Wunsch
   All rights reserved.

   Portions of documentation Copyright (c) 1991, 1993
   The Regents of the University of California.

   All rights reserved.

   Redistribution and use in source and binary forms, with or without
   modification, are permitted provided that the following conditions are met:

   * Redistributions of source code must retain the above copyright
     notice, this list of conditions and the following disclaimer.

   * Redistributions in binary form must reproduce the above copyright
     notice, this list of conditions and the following disclaimer in
     the documentation and/or other materials provided with the
     distribution.

   * Neither the name of the copyright holders nor the names of
     contributors may be used to endorse or promote products derived
     from this software without specific prior written permission.

  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
  AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
  IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
  ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
  LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
  CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
  SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
  INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
  CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
  ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
  POSSIBILITY OF SUCH DAMAGE.

  $Id$
*/

/** \file */
/** \defgroup avr_assert <assert.h>: Diagnostics
    \code #include <assert.h> \endcode

    This header file defines a debugging aid.

    As there is no standard error output stream available for many
    applications using this library, the generation of a printable
    error message is not enabled by default.  These messages will
    only be generated if the application defines the macro

    \code __ASSERT_USE_STDERR \endcode

    before including the \c <assert.h> header file.  By default,
    only abort() will be called to halt the application.
*/

/**@{*/

/*
 * The ability to include this file (with or without NDEBUG) is a
 * feature.
 */

#undef assert

#include <stdlib.h>

#if defined(__DOXYGEN__)
/**
 * \def assert
 * \param expression Expression to test for.
```

```
 *
 * The assert() macro tests the given expression and if it is false,
 * the calling process is terminated.  A diagnostic message is written
 * to stderr and the function abort() is called, effectively
 * terminating the program.
 *
 * If expression is true, the assert() macro does nothing.
 *
 * The assert() macro may be removed at compile time by defining
 * NDEBUG as a macro (e.g., by using the compiler option -DNDEBUG).
 */
#  define assert(expression)

#else /* !DOXYGEN */

#  if defined(NDEBUG)
#    define assert(e)    ((void)0)
#  else /* !NDEBUG */
#    if defined(__ASSERT_USE_STDERR)
#      define assert(e) ((e) ? (void)0 : \
                        __assert(__func__, __FILE__, __LINE__, #e))
#    else /* !__ASSERT_USE_STDERR */
#      define assert(e) ((e) ? (void)0 : abort())
#    endif /* __ASSERT_USE_STDERR */
#  endif /* NDEBUG */
#endif /* DOXYGEN */

#if (defined __STDC_VERSION__ && __STDC_VERSION__ >= 201112L) || \
    ((_GNUC_ > 4 || (_GNUC_ == 4 && _GNUC_MINOR_ >= 6)) && !defined __cplusplus)
#  undef static_assert
#  define static_assert _Static_assert
#endif

#ifdef __cplusplus
extern "C" {
#endif

#if !defined(__DOXYGEN__)

extern void __assert(const char *__func, const char *__file,
                     int __lineno, const char *__sexp);

#endif /* not __DOXYGEN__ */

#ifdef __cplusplus
}
#endif

/**@}*/
/* EOF */
```

23.10 boot.h File Reference

Macros

- #define BOOTLOADER_SECTION __attribute__ ((__section__(".bootloader")))
- #define boot_spm_interrupt_enable() (__SPM_REG |= (uint8_t)_BV(SPMIE))
- #define boot_spm_interrupt_disable() (__SPM_REG &= (uint8_t)~_BV(SPMIE))
- #define boot_is_spm_interrupt() (__SPM_REG & (uint8_t)_BV(SPMIE))
- #define boot_rww_busy() (__SPM_REG & (uint8_t)_BV(__COMMON_ASB))
- #define boot_spm_busy() (__SPM_REG & (uint8_t)_BV(__SPM_ENABLE))
- #define boot_spm_busy_wait() do{}while(boot_spm_busy())
- #define GET_LOW_FUSE_BITS (0x0000)
- #define GET_LOCK_BITS (0x0001)
- #define GET_EXTENDED_FUSE_BITS (0x0002)
- #define GET_HIGH_FUSE_BITS (0x0003)
- #define boot_lock_fuse_bits_get(address)
- #define boot_signature_byte_get(addr)
- #define boot_page_fill(address, data) __boot_page_fill_normal(address, data)
- #define boot_page_erase(address) __boot_page_erase_normal(address)
- #define boot_page_write(address) __boot_page_write_normal(address)
- #define boot_rww_enable() __boot_rww_enable()

- #define boot_lock_bits_set(lock_bits) __boot_lock_bits_set(lock_bits)
- #define boot_page_fill_safe(address, data)
- #define boot_page_erase_safe(address)
- #define boot_page_write_safe(address)
- #define boot_rww_enable_safe()
- #define boot_lock_bits_set_safe(lock_bits)

23.11 boot.h

Go to the documentation of this file.

```
/* Copyright (c) 2002,2003,2004,2005,2006,2007,2008,2009  Eric B. Weddington
   All rights reserved.

   Redistribution and use in source and binary forms, with or without
   modification, are permitted provided that the following conditions are met:

   * Redistributions of source code must retain the above copyright
     notice, this list of conditions and the following disclaimer.
   * Redistributions in binary form must reproduce the above copyright
     notice, this list of conditions and the following disclaimer in
     the documentation and/or other materials provided with the
     distribution.
   * Neither the name of the copyright holders nor the names of
     contributors may be used to endorse or promote products derived
     from this software without specific prior written permission.

  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
  AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
  IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
  ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
  LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
  CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
  SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
  INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
  CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
  ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
  POSSIBILITY OF SUCH DAMAGE. */

/* $Id$ */

#ifndef _AVR_BOOT_H_
#define _AVR_BOOT_H_    1

/** \file */
/** \defgroup avr_boot <avr/boot.h>: Bootloader Support Utilities
    \code
    #include <avr/io.h>
    #include <avr/boot.h>
    \endcode

    The macros in this module provide a C language interface to the
    bootloader support functionality of certain AVR processors. These
    macros are designed to work with all sizes of flash memory.

    Global interrupts are not automatically disabled for these macros. It
    is left up to the programmer to do this. See the code example below.
    Also see the processor datasheet for caveats on having global interrupts
    enabled during writing of the Flash.

    \note Not all AVR processors provide bootloader support. See your
    processor datasheet to see if it provides bootloader support.

    \par API Usage Example
    The following code shows typical usage of the boot API.

    \code
    #include <stdint.h>
    #include <avr/interrupt.h>
    #include <avr/pgmspace.h>

    void boot_program_page (uint32_t page, uint8_t *buf)
    {
        // Disable interrupts.
        uint8_t sreg = SREG;
        cli();

        eeprom_busy_wait ();

        boot_page_erase (page);
```

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

        for (uint16_t i = 0; i < SPM_PAGESIZE; i += 2)
        {
            // Set up little-endian word.
            uint16_t w = *buf++;
            w += (*buf++) « 8;

            boot_page_fill (page + i, w);
        }

        boot_page_write (page);     // Store buffer in flash page.
        boot_spm_busy_wait();       // Wait until the memory is written.

        // Reenable RWW-section again. We need this if we want to jump back
        // to the application after bootloading.
        boot_rww_enable ();

        // Re-enable interrupts (if they were ever enabled).
        SREG = sreg;
    }\endcode */

#include <avr/eeprom.h>
#include <avr/io.h>
#include <inttypes.h>
#include <limits.h>

/* Check for SPM Control Register in processor. */
#if defined (SPMCSR)
#  define __SPM_REG    SPMCSR
#else
#  if defined (SPMCR)
#    define __SPM_REG    SPMCR
#  else
#    error AVR processor does not provide bootloader support!
#  endif
#endif


/* Check for SPM Enable bit. */
#if defined(SPMEN)
#  define __SPM_ENABLE  SPMEN
#elif defined(SELFPRGEN)
#  define __SPM_ENABLE  SELFPRGEN
#else
#  error Cannot find SPM Enable bit definition!
#endif

/** \ingroup avr_boot
    \def BOOTLOADER_SECTION

    Used to declare a function or variable to be placed into a
    new section called .bootloader. This section and its contents
    can then be relocated to any address (such as the bootloader
    NRWW area) at link-time. */

#define BOOTLOADER_SECTION    __attribute__ ((__section__(".bootloader")))

#ifndef __DOXYGEN__
/* Create common bit definitions. */
#ifdef ASB
#define __COMMON_ASB    ASB
#else
#define __COMMON_ASB    RWWSB
#endif

#ifdef ASRE
#define __COMMON_ASRE   ASRE
#else
#define __COMMON_ASRE   RWWSRE
#endif

/* Define the bit positions of the Boot Lock Bits. */

#define BLB12           5
#define BLB11           4
#define BLB02           3
#define BLB01           2
#endif  /* __DOXYGEN__ */

/** \ingroup avr_boot
    \def boot_spm_interrupt_enable()
    Enable the SPM interrupt. */

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

/** \ingroup avr_boot
```

```
    \def boot_spm_interrupt_disable()
    Disable the SPM interrupt. */

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

/** \ingroup avr_boot
    \def boot_is_spm_interrupt()
    Check if the SPM interrupt is enabled. */

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

/** \ingroup avr_boot
    \def boot_rww_busy()
    Check if the RWW section is busy. */

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

/** \ingroup avr_boot
    \def boot_spm_busy()
    Check if the SPM instruction is busy. */

#define boot_spm_busy()               (__SPM_REG & (uint8_t)_BV(__SPM_ENABLE))

/** \ingroup avr_boot
    \def boot_spm_busy_wait()
    Wait while the SPM instruction is busy. */

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

#ifndef __DOXYGEN__
#define __BOOT_PAGE_ERASE         (_BV(__SPM_ENABLE) | _BV(PGERS))
#define __BOOT_PAGE_WRITE         (_BV(__SPM_ENABLE) | _BV(PGWRT))
#define __BOOT_PAGE_FILL          _BV(__SPM_ENABLE)
#define __BOOT_RWW_ENABLE         (_BV(__SPM_ENABLE) | _BV(__COMMON_ASRE))
#if defined(BLBSET)
#define __BOOT_LOCK_BITS_SET      (_BV(__SPM_ENABLE) | _BV(BLBSET))
#elif defined(RFLB)  /* Some devices have RFLB defined instead of BLBSET. */
#define __BOOT_LOCK_BITS_SET      (_BV(__SPM_ENABLE) | _BV(RFLB))
#elif defined(RWFLB)  /* Some devices have RWFLB defined instead of BLBSET. */
#define __BOOT_LOCK_BITS_SET      (_BV(__SPM_ENABLE) | _BV(RWFLB))
#endif

#define __boot_page_fill_normal(address, data)   \
(__extension__({                                 \
    if (_SFR_IO_REG_P(__SPM_REG))                \
      __asm__ __volatile__ (                     \
        "movw r0, %3"     "\n\t"                 \
        "out  %0, %1"     "\n\t"                 \
        "spm"             "\n\t"                 \
        "clr  __zero_reg__"                      \
        :                                        \
        : "i" (_SFR_IO_ADDR(__SPM_REG)),         \
          "r" ((uint8_t)(__BOOT_PAGE_FILL)),     \
          "z" ((uint16_t)(address)),             \
          "r" ((uint16_t)(data))                 \
        : "r0");                                 \
    else                                         \
      __asm__ __volatile__ (                     \
        "movw r0, %3"      "\n\t"                \
        "sts  %0, %1"      "\n\t"                \
        "spm"              "\n\t"                \
        "clr  __zero_reg__"                      \
        :                                        \
        : "i" (_SFR_MEM_ADDR(__SPM_REG)),        \
          "r" ((uint8_t)(__BOOT_PAGE_FILL)),     \
          "z" ((uint16_t)(address)),             \
          "r" ((uint16_t)(data))                 \
        : "r0");                                 \
}))

#define __boot_page_fill_alternate(address, data)\
(__extension__({                                 \
    __asm__ __volatile__                         \
    (                                            \
        "movw r0, %3"     "\n\t"                 \
        "sts  %0, %1"     "\n\t"                 \
        "spm"             "\n\t"                 \
        ".word 0xffff"    "\n\t"                 \
        "nop"             "\n\t"                 \
        "clr  __zero_reg__"                      \
        :                                        \
        : "i" (_SFR_MEM_ADDR(__SPM_REG)),        \
          "r" ((uint8_t)(__BOOT_PAGE_FILL)),     \
          "z" ((uint16_t)(address)),             \
          "r" ((uint16_t)(data))                 \
        : "r0"                                   \
    );                                           \
```

```
}))

#define __boot_page_fill_extended(address, data) \
(__extension__({                                 \
    __asm__ __volatile__                         \
    (                                            \
        "movw r0, %4"      "\n\t"                \
        "movw r30, %A3"    "\n\t"                \
        "out  %1, %C3"     "\n\t"                \
        "sts  %0, %2"      "\n\t"                \
        "spm"              "\n\t"                \
        "clr __zero_reg__"                       \
        :                                        \
        : "i" (_SFR_MEM_ADDR(__SPM_REG)),        \
          "i" (_SFR_IO_ADDR(RAMPZ)),             \
          "r" ((uint8_t)(__BOOT_PAGE_FILL)),     \
          "r" ((uint32_t)(address)),             \
          "r" ((uint16_t)(data))                 \
        : "r0", "r30", "r31"                     \
    );                                           \
}))

#define __boot_page_erase_normal(address)        \
(__extension__({                                 \
    if (_SFR_IO_REG_P(__SPM_REG))                \
      __asm__ __volatile__ (                     \
        "out %0, %1"       "\n\t"                \
        "spm"                                    \
        :                                        \
        : "i" (_SFR_IO_ADDR(__SPM_REG)),         \
          "r" ((uint8_t)(__BOOT_PAGE_ERASE)),    \
          "z" ((uint16_t)(address)));            \
    else                                         \
      __asm__ __volatile__ (                     \
        "sts %0, %1"       "\n\t"                \
        "spm"                                    \
        :                                        \
        : "i" (_SFR_MEM_ADDR(__SPM_REG)),        \
          "r" ((uint8_t)(__BOOT_PAGE_ERASE)),    \
          "z" ((uint16_t)(address)));            \
}))

#define __boot_page_erase_alternate(address)     \
(__extension__({                                 \
    __asm__ __volatile__                         \
    (                                            \
        "sts %0, %1"      "\n\t"                 \
        "spm"             "\n\t"                 \
        ".word 0xffff"    "\n\t"                 \
        "nop"                                    \
        :                                        \
        : "i" (_SFR_MEM_ADDR(__SPM_REG)),        \
          "r" ((uint8_t)(__BOOT_PAGE_ERASE)),    \
          "z" ((uint16_t)(address))              \
    );                                           \
}))

#define __boot_page_erase_extended(address)      \
(__extension__({                                 \
    __asm__ __volatile__                         \
    (                                            \
        "movw r30, %A3"     "\n\t"               \
        "out  %1, %C3"      "\n\t"               \
        "sts  %0, %2"       "\n\t"               \
        "spm"                                    \
        :                                        \
        : "i" (_SFR_MEM_ADDR(__SPM_REG)),        \
          "i" (_SFR_IO_ADDR(RAMPZ)),             \
          "r" ((uint8_t)(__BOOT_PAGE_ERASE)),    \
          "r" ((uint32_t)(address))              \
        : "r30", "r31"                           \
    );                                           \
}))

#define __boot_page_write_normal(address)        \
(__extension__({                                 \
    if (_SFR_IO_REG_P(__SPM_REG))                \
      __asm__ __volatile__ (                     \
        "out %0, %1"       "\n\t"                \
        "spm"                                    \
        :                                        \
        : "i" (_SFR_IO_ADDR(__SPM_REG)),         \
          "r" ((uint8_t)(__BOOT_PAGE_WRITE)),    \
          "z" ((uint16_t)(address)));            \
    else                                         \
      __asm__ __volatile__ (                     \
        "sts %0, %1"       "\n\t"                \
```

```
        "spm"                                  \
        :                                      \
        : "i" (_SFR_MEM_ADDR(__SPM_REG)),      \
          "r" ((uint8_t)(__BOOT_PAGE_WRITE)),  \
          "z" ((uint16_t)(address)));          \
}))

#define __boot_page_write_alternate(address)   \
(__extension__({                               \
    __asm__ __volatile__                       \
    (                                          \
        "sts %0, %1"        "\n\t"             \
        "spm"               "\n\t"             \
        ".word 0xffff"      "\n\t"             \
        "nop"                                  \
        :                                      \
        : "i" (_SFR_MEM_ADDR(__SPM_REG)),      \
          "r" ((uint8_t)(__BOOT_PAGE_WRITE)),  \
          "z" ((uint16_t)(address))            \
    );                                         \
}))

#define __boot_page_write_extended(address)    \
(__extension__({                               \
    __asm__ __volatile__                       \
    (                                          \
        "movw r30, %A3"       "\n\t"           \
        "out  %1, %C3"        "\n\t"           \
        "sts  %0, %2"         "\n\t"           \
        "spm"                                  \
        :                                      \
        : "i" (_SFR_MEM_ADDR(__SPM_REG)),      \
          "i" (_SFR_IO_ADDR(RAMPZ)),           \
          "r" ((uint8_t)(__BOOT_PAGE_WRITE)),  \
          "r" ((uint32_t)(address))            \
        : "r30", "r31"                         \
    );                                         \
}))

#define __boot_rww_enable()                    \
(__extension__({                               \
    __asm__ __volatile__                       \
    (                                          \
        "sts %0, %1"      "\n\t"               \
        "spm"                                  \
        :                                      \
        : "i" (_SFR_MEM_ADDR(__SPM_REG)),      \
          "r" ((uint8_t)(__BOOT_RWW_ENABLE))   \
    );                                         \
}))

#define __boot_rww_enable_alternate()          \
(__extension__({                               \
    __asm__ __volatile__                       \
    (                                          \
        "sts %0, %1"        "\n\t"             \
        "spm"               "\n\t"             \
        ".word 0xffff"      "\n\t"             \
        "nop"                                  \
        :                                      \
        : "i" (_SFR_MEM_ADDR(__SPM_REG)),      \
          "r" ((uint8_t)(__BOOT_RWW_ENABLE))   \
    );                                         \
}))

/* From the mega16/mega128 data sheets (maybe others):

     Bits by SPM To set the Boot Loader Lock bits, write the desired data to
     R0, write "X0001001" to SPMCR and execute SPM within four clock cycles
     after writing SPMCR. The only accessible Lock bits are the Boot Lock bits
     that may prevent the Application and Boot Loader section from any
     software update by the MCU.

     If bits 5..2 in R0 are cleared (zero), the corresponding Boot Lock bit
     will be programmed if an SPM instruction is executed within four cycles
     after BLBSET and SPMEN (or SELFPRGEN) are set in SPMCR. The Z-pointer is
     don't care during this operation, but for future compatibility it is
     recommended to load the Z-pointer with $0001 (same as used for reading the
     Lock bits). For future compatibility It is also recommended to set bits 7,
     6, 1, and 0 in R0 to 1 when writing the Lock bits. When programming the
     Lock bits the entire Flash can be read during the operation. */

#define __boot_lock_bits_set(lock_bits)                    \
(__extension__({                                           \
    uint8_t value = (uint8_t)(~(lock_bits));               \
    __asm__ __volatile__                                   \
    (                                                      \
```

```
        "ldi r30, 1"      "\n\t"                                  \
        "ldi r31, 0"      "\n\t"                                  \
        "mov r0, %2"      "\n\t"                                  \
        "sts %0, %1"      "\n\t"                                  \
        "spm"                                                     \
        :                                                         \
        : "i" (_SFR_MEM_ADDR(__SPM_REG)),                         \
          "r" ((uint8_t)(__BOOT_LOCK_BITS_SET)),                  \
          "r" (value)                                             \
        : "r0", "r30", "r31"                                      \
    );                                                            \
}))

#define __boot_lock_bits_set_alternate(lock_bits)                 \
(__extension__({                                                  \
    uint8_t value = (uint8_t)(~(lock_bits));                      \
    __asm__ __volatile__                                          \
    (                                                             \
        "ldi r30, 1"     "\n\t"                                   \
        "ldi r31, 0"     "\n\t"                                   \
        "mov r0, %2"     "\n\t"                                   \
        "sts %0, %1"     "\n\t"                                   \
        "spm"            "\n\t"                                   \
        ".word 0xffff"   "\n\t"                                   \
        "nop"                                                     \
        :                                                         \
        : "i" (_SFR_MEM_ADDR(__SPM_REG)),                         \
          "r" ((uint8_t)(__BOOT_LOCK_BITS_SET)),                  \
          "r" (value)                                             \
        : "r0", "r30", "r31"                                      \
    );                                                            \
}))
#endif  /* __DOXYGEN__ */

/*
   Reading lock and fuse bits:

     Similarly to writing the lock bits above, set BLBSET and SPMEN (or
     SELFPRGEN) bits in __SPMREG, and then (within four clock cycles) issue an
     LPM instruction.

     Z address:       contents:
     0x0000           low fuse bits
     0x0001           lock bits
     0x0002           extended fuse bits
     0x0003           high fuse bits

     Sounds confusing, doesn't it?

     Unlike the macros in pgmspace.h, no need to care for non-enhanced
     cores here as these old cores do not provide SPM support anyway.
 */

/** \ingroup avr_boot
    \def GET_LOW_FUSE_BITS
    address to read the low fuse bits, using boot_lock_fuse_bits_get
 */
#define GET_LOW_FUSE_BITS           (0x0000)
/** \ingroup avr_boot
    \def GET_LOCK_BITS
    address to read the lock bits, using boot_lock_fuse_bits_get
 */
#define GET_LOCK_BITS               (0x0001)
/** \ingroup avr_boot
    \def GET_EXTENDED_FUSE_BITS
    address to read the extended fuse bits, using boot_lock_fuse_bits_get
 */
#define GET_EXTENDED_FUSE_BITS      (0x0002)
/** \ingroup avr_boot
    \def GET_HIGH_FUSE_BITS
    address to read the high fuse bits, using boot_lock_fuse_bits_get
 */
#define GET_HIGH_FUSE_BITS          (0x0003)

/** \ingroup avr_boot
    \def boot_lock_fuse_bits_get(address)

    Read the lock or fuse bits at \c address.

    Parameter \c address can be any of GET_LOW_FUSE_BITS,
    GET_LOCK_BITS, GET_EXTENDED_FUSE_BITS, or GET_HIGH_FUSE_BITS.

    \note The lock and fuse bits returned are the physical values,
    i.e. a bit returned as 0 means the corresponding fuse or lock bit
    is programmed.
 */
#define boot_lock_fuse_bits_get(address)                          \
```

```
(__extension__({                                              \
    uint8_t __result;                                         \
    __asm__ __volatile__                                      \
    (                                                         \
        "sts %1, %2\n\t"                                      \
        "lpm %0, Z\n\t"                                       \
        : "=r" (__result)                                     \
        : "i" (_SFR_MEM_ADDR(__SPM_REG)),                     \
          "r" ((uint8_t)(__BOOT_LOCK_BITS_SET)),              \
          "z" ((uint16_t)(address))                           \
    );                                                        \
    __result;                                                 \
}))

#ifndef __DOXYGEN__
#  if defined(SIGRD)
#    define __BOOT_SIGROW_READ (_BV(__SPM_ENABLE) | _BV(SIGRD))
#  elif defined(RSIG)
#    define __BOOT_SIGROW_READ (_BV(__SPM_ENABLE) | _BV(RSIG))
#  endif
#endif

/** \ingroup avr_boot
    \def boot_signature_byte_get(address)

    Read the Signature Row byte at \c address.  For some MCU types,
    this function can also retrieve the factory-stored oscillator
    calibration bytes.

    Parameter \c address can be 0-0x1f as documented by the datasheet.
    \note The values are MCU type dependent.
*/

#define boot_signature_byte_get(addr)                \
  (__extension__({                                   \
      uint8_t __result;                              \
      __asm__ __volatile__                           \
      (                                              \
        "sts %1, %2"   "\n\t"                        \
        "lpm %0, Z"                                  \
        : "=r" (__result)                            \
        : "i" (_SFR_MEM_ADDR(__SPM_REG)),            \
          "r" ((uint8_t)(__BOOT_SIGROW_READ)),       \
          "z" ((uint16_t)(addr))                     \
      );                                             \
      __result;                                      \
}))

/** \ingroup avr_boot
    \def boot_page_fill(address, data)

    Fill the bootloader temporary page buffer for flash
    address with data word.

    \note The address is a byte address. The data is a word. The AVR
    writes data to the buffer a word at a time, but addresses the buffer
    per byte! So, increment your address by 2 between calls, and send 2
    data bytes in a word format! The LSB of the data is written to the lower
    address; the MSB of the data is written to the higher address.*/

/** \ingroup avr_boot
    \def boot_page_erase(address)

    Erase the flash page that contains address.

    \note address is a byte address in flash, not a word address. */

/** \ingroup avr_boot
    \def boot_page_write(address)

    Write the bootloader temporary page buffer
    to flash page that contains address.

    \note address is a byte address in flash, not a word address. */

/** \ingroup avr_boot
    \def boot_rww_enable()

    Enable the Read-While-Write memory section. */

/** \ingroup avr_boot
    \def boot_lock_bits_set(lock_bits)

    Set the bootloader lock bits.

    \param lock_bits A mask of which Boot Loader Lock Bits to set.

```

```
    \note In this context, a 'set bit' will be written to a zero value.
    Note also that only BLBxx bits can be programmed by this command.

    For example, to disallow the SPM instruction from writing to the Boot
    Loader memory section of flash, you would use this macro as such:

    \code
    boot_lock_bits_set (_BV (BLB11));
    \endcode

    \note Like any lock bits, the Boot Loader Lock Bits, once set,
    cannot be cleared again except by a chip erase which will in turn
    also erase the boot loader itself. */

/* Normal versions of the macros use 16-bit addresses.
   Extended versions of the macros use 32-bit addresses.
   Alternate versions of the macros use 16-bit addresses and require special
   instruction sequences after LPM.

   FLASHEND is defined in the ioXXXX.h file.
   USHRT_MAX is defined in <limits.h>. */

#if defined(__AVR_ATmega161__) || defined(__AVR_ATmega163__) \
    || defined(__AVR_ATmega323__)

/* Alternate: ATmega161/163/323 and 16 bit address */
#define boot_page_fill(address, data) __boot_page_fill_alternate(address, data)
#define boot_page_erase(address)      __boot_page_erase_alternate(address)
#define boot_page_write(address)      __boot_page_write_alternate(address)
#define boot_rww_enable()             __boot_rww_enable_alternate()
#define boot_lock_bits_set(lock_bits) __boot_lock_bits_set_alternate(lock_bits)

#elif (FLASHEND > USHRT_MAX)

/* Extended: >16 bit address */
#define boot_page_fill(address, data) __boot_page_fill_extended(address, data)
#define boot_page_erase(address)      __boot_page_erase_extended(address)
#define boot_page_write(address)      __boot_page_write_extended(address)
#define boot_rww_enable()             __boot_rww_enable()
#define boot_lock_bits_set(lock_bits) __boot_lock_bits_set(lock_bits)

#else

/* Normal: 16 bit address */
#define boot_page_fill(address, data) __boot_page_fill_normal(address, data)
#define boot_page_erase(address)      __boot_page_erase_normal(address)
#define boot_page_write(address)      __boot_page_write_normal(address)
#define boot_rww_enable()             __boot_rww_enable()
#define boot_lock_bits_set(lock_bits) __boot_lock_bits_set(lock_bits)

#endif

/** \ingroup avr_boot

    Same as boot_page_fill() except it waits for eeprom and spm operations to
    complete before filling the page. */

#define boot_page_fill_safe(address, data) \
do { \
    boot_spm_busy_wait();                       \
    eeprom_busy_wait();                         \
    boot_page_fill(address, data);              \
} while (0)

/** \ingroup avr_boot

    Same as boot_page_erase() except it waits for eeprom and spm operations to
    complete before erasing the page. */

#define boot_page_erase_safe(address) \
do { \
    boot_spm_busy_wait();                       \
    eeprom_busy_wait();                         \
    boot_page_erase (address);                  \
} while (0)

/** \ingroup avr_boot

    Same as boot_page_write() except it waits for eeprom and spm operations to
    complete before writing the page. */

#define boot_page_write_safe(address) \
do { \
    boot_spm_busy_wait();                       \
    eeprom_busy_wait();                         \
    boot_page_write (address);                  \
} while (0)
```

```

/** \ingroup avr_boot

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

#define boot_rww_enable_safe() \
do { \
    boot_spm_busy_wait();                        \
    eeprom_busy_wait();                          \
    boot_rww_enable();                           \
} while (0)

/** \ingroup avr_boot

    Same as boot_lock_bits_set() except waits for eeprom and spm operations to
    complete before setting the lock bits. */

#define boot_lock_bits_set_safe(lock_bits) \
do { \
    boot_spm_busy_wait();                        \
    eeprom_busy_wait();                          \
    boot_lock_bits_set (lock_bits);              \
} while (0)

#endif /* _AVR_BOOT_H_ */
```

23.12 builtins.h File Reference

Macros

- #define **__HAS_DELAY_CYCLES** 1

Functions

- void __builtin_avr_sei (void)
- void __builtin_avr_cli (void)
- void __builtin_avr_sleep (void)
- void __builtin_avr_wdr (void)
- uint8_t __builtin_avr_swap (uint8_t __b)
- uint16_t __builtin_avr_fmul (uint8_t __a, uint8_t __b)
- int16_t __builtin_avr_fmuls (int8_t __a, int8_t __b)
- int16_t __builtin_avr_fmulsu (int8_t __a, uint8_t __b)

23.13 builtins.h

Go to the documentation of this file.

```
/* Copyright (c) 2008 Anatoly Sokolov
   Copyright (c) 2010 Joerg Wunsch
   All rights reserved.

   Redistribution and use in source and binary forms, with or without
   modification, are permitted provided that the following conditions are met:

   * Redistributions of source code must retain the above copyright
     notice, this list of conditions and the following disclaimer.

   * Redistributions in binary form must reproduce the above copyright
     notice, this list of conditions and the following disclaimer in
     the documentation and/or other materials provided with the
     distribution.

   * Neither the name of the copyright holders nor the names of
     contributors may be used to endorse or promote products derived
     from this software without specific prior written permission.

  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
  AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
```

```
 IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
 LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
 CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
 SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
 INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
 CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
 ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
 POSSIBILITY OF SUCH DAMAGE. */

/* $Id$ */

/*
   avr/builtins.h - Intrinsic functions built into the compiler
 */

#ifndef _AVR_BUILTINS_H_
#define _AVR_BUILTINS_H_

#ifndef __HAS_DELAY_CYCLES
#define __HAS_DELAY_CYCLES 1
#endif

/* For GCC built-ins, we should not define prototypes,
   hence only document that stuff.  */
#ifdef __DOXYGEN__

/** \file */
/** \defgroup avr_builtins <avr/builtins.h>: avr-gcc builtins documentation
    \code #include <avr/builtins.h> \endcode

    \note This file only documents some avr-gcc builtins.
    For functions built-in in the compiler, there should be no
    prototype declarations.

    See also the
    <a href="https://gcc.gnu.org/onlinedocs/gcc/AVR-Built-in-Functions.html"
       >GCC documentation</a> for a full list of avr-gcc builtins.
*/

/**
    \ingroup avr_builtins

    Enables interrupts by setting the global interrupt mask.  */
extern void __builtin_avr_sei(void);

/**
    \ingroup avr_builtins

    Disables all interrupts by clearing the global interrupt mask.  */
extern void __builtin_avr_cli(void);

/**
    \ingroup avr_builtins

    Emits a \c SLEEP instruction.  */

extern void __builtin_avr_sleep(void);

/**
    \ingroup avr_builtins

    Emits a WDR (watchdog reset) instruction.  */
extern void __builtin_avr_wdr(void);

/**
    \ingroup avr_builtins

    Emits a SWAP (nibble swap) instruction on __b.  */
extern uint8_t __builtin_avr_swap(uint8_t __b);

/**
    \ingroup avr_builtins

    Emits an FMUL (fractional multiply unsigned) instruction.  */
extern uint16_t __builtin_avr_fmul(uint8_t __a, uint8_t __b);

/**
    \ingroup avr_builtins

    Emits an FMUL (fractional multiply signed) instruction.  */
extern int16_t __builtin_avr_fmuls(int8_t __a, int8_t __b);

/**
    \ingroup avr_builtins

    Emits an FMUL (fractional multiply signed with unsigned) instruction.  */
```

```
extern int16_t __builtin_avr_fmulsu(int8_t __a, uint8_t __b);

#if __HAS_DELAY_CYCLES
/**
    \ingroup avr_builtins

    Emits a sequence of instructions causing the CPU to spend
    \c __n cycles on it.  */
extern void __builtin_avr_delay_cycles(uint32_t __n);
#endif
#endif /* DOXYGEN */
#endif /* _AVR_BUILTINS_H_ */
```

23.14 cpufunc.h File Reference

Macros

- #define _NOP()
- #define _MemoryBarrier()

Functions

- void ccp_write_io (volatile uint8_t ∗__ioaddr, uint8_t __value)
- void ccp_write_spm (volatile uint8_t ∗__ioaddr, uint8_t __value)

23.15 cpufunc.h

Go to the documentation of this file.

```
/* Copyright (c) 2010, Joerg Wunsch
   All rights reserved.

   Redistribution and use in source and binary forms, with or without
   modification, are permitted provided that the following conditions are met:

   * Redistributions of source code must retain the above copyright
     notice, this list of conditions and the following disclaimer.

   * Redistributions in binary form must reproduce the above copyright
     notice, this list of conditions and the following disclaimer in
     the documentation and/or other materials provided with the
     distribution.

   * Neither the name of the copyright holders nor the names of
     contributors may be used to endorse or promote products derived
     from this software without specific prior written permission.

  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
  AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
  IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
  ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
  LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
  CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
  SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
  INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
  CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
  ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
  POSSIBILITY OF SUCH DAMAGE. */

/* $Id$ */

/* avr/cpufunc.h - Special CPU functions */

#ifndef _AVR_CPUFUNC_H_
#define _AVR_CPUFUNC_H_ 1

#include <stdint.h>

/** \file */
/** \defgroup avr_cpufunc <avr/cpufunc.h>: Special AVR CPU functions
    \code #include <avr/cpufunc.h> \endcode

```

```
     This header file contains macros that access special functions of
     the AVR CPU which do not fit into any of the other header files.

*/

#if defined(__DOXYGEN__)
/**
   \ingroup avr_cpufunc
   \def _NOP

   Execute a <i>no operation</i> (NOP) CPU instruction.  This
   should not be used to implement delays, better use the functions
   from <util/delay_basic.h> or <util/delay.h> for this.  For
   debugging purposes, a NOP can be useful to have an instruction that
   is guaranteed to be not optimized away by the compiler, so it can
   always become a breakpoint in the debugger.
*/
#define _NOP()
#else  /* real code */
#define _NOP() __asm__ __volatile__("nop")
#endif  /* __DOXYGEN__ */

#if defined(__DOXYGEN__)
/**
   \ingroup avr_cpufunc
   \def _MemoryBarrier

   Implement a read/write <i>memory barrier</i>.  A memory
   barrier instructs the compiler to not cache any memory data in
   registers beyond the barrier.  This can sometimes be more effective
   than blocking certain optimizations by declaring some object with a
   \c volatile qualifier.

   See \ref optim_code_reorder for things to be taken into account
   with respect to compiler optimizations.
*/
#define _MemoryBarrier()
#else  /* real code */
#define _MemoryBarrier() __asm__ __volatile__("":::"memory")
#endif  /* __DOXYGEN__ */

#ifdef __cplusplus
extern "C" {
#endif

/**
   \ingroup avr_cpufunc

   Write \a __value to IO Register Protected (CCP) IO register
   at \a __ioaddr.   . See also \c _PROTECTED_WRITE().  */
void ccp_write_io (volatile uint8_t *__ioaddr, uint8_t __value);

/**
   \ingroup avr_cpufunc

   Write \a __value to SPM Instruction Protected (CCP) IO register
   at \a __ioaddr. See also \c _PROTECTED_WRITE_SPM().  */
void ccp_write_spm (volatile uint8_t *__ioaddr, uint8_t __value);

#ifdef __cplusplus
}
#endif

#endif /* _AVR_CPUFUNC_H_ */
```

23.16 eeprom.h

```
/* Copyright (c) 2002, 2003, 2004, 2007 Marek Michalkiewicz
   Copyright (c) 2005, 2006 Bjoern Haase
   Copyright (c) 2008 Atmel Corporation
   Copyright (c) 2008 Wouter van Gulik
   Copyright (c) 2009 Dmitry Xmelkov
   All rights reserved.

   Redistribution and use in source and binary forms, with or without
   modification, are permitted provided that the following conditions are met:

   * Redistributions of source code must retain the above copyright
     notice, this list of conditions and the following disclaimer.
   * Redistributions in binary form must reproduce the above copyright
     notice, this list of conditions and the following disclaimer in
     the documentation and/or other materials provided with the
     distribution.
   * Neither the name of the copyright holders nor the names of
```

```
     contributors may be used to endorse or promote products derived
     from this software without specific prior written permission.

  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
  AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
  IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
  ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
  LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
  CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
  SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
  INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
  CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
  ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
  POSSIBILITY OF SUCH DAMAGE. */

/* $Id$ */

#ifndef _AVR_EEPROM_H_
#define _AVR_EEPROM_H_ 1

#include <avr/io.h>

#if !E2END && !defined(__DOXYGEN__) && !defined(__COMPILING_AVR_LIBC__)
# warning "Device does not have EEPROM available."
#else

#if defined (EEAR) && !defined (EEARL) && !defined (EEARH)
#define EEARL EEAR
#endif

#ifndef __ASSEMBLER__

#include <stddef.h> /* size_t */
#include <stdint.h>

/** \defgroup avr_eeprom <avr/eeprom.h>: EEPROM handling
    \code #include <avr/eeprom.h> \endcode

    This header file declares the interface to some simple library
    routines suitable for handling the data EEPROM contained in the
    AVR microcontrollers.  The implementation uses a simple polled
    mode interface.  Applications that require interrupt-controlled
    EEPROM access to ensure that no time will be wasted in spinloops
    will have to deploy their own implementation.

    \par Notes:

    - In addition to the write functions there is a set of update ones.
    This functions read each byte first and skip the burning if the
    old value is the same with new.  The scaning direction is from
    high address to low, to obtain quick return in common cases.

    - All of the read/write functions first make sure the EEPROM is
    ready to be accessed.  Since this may cause long delays if a
    write operation is still pending, time-critical applications
    should first poll the EEPROM e. g. using eeprom_is_ready() before
    attempting any actual I/O.  But this functions does not wait until
    SELFPRGEN in SPMCSR becomes zero.  Do this manually, if your
    softwate contains the Flash burning.

    - As these functions modify IO registers, they are known to be
    non-reentrant.  If any of these functions are used from both,
    standard and interrupt context, the applications must ensure
    proper protection (e.g. by disabling interrupts before accessing
    them).

    - All write functions force erase_and_write programming mode.

    - For Xmega the EEPROM start address is 0, like other architectures.
    The reading functions add the 0x2000 value to use EEPROM mapping into
    data space.
 */

#ifdef __cplusplus
extern "C" {
#endif

#ifndef __ATTR_PURE__
# ifdef  __DOXYGEN__
#  define __ATTR_PURE__
# else
#  define __ATTR_PURE__  __attribute__((__pure__))
# endif
#endif

/** \def EEMEM
    \ingroup avr_eeprom
```

```
    Attribute expression causing a variable to be allocated within the
    .eeprom section.    */
#define EEMEM __attribute__((__section__(".eeprom")))

/** \def eeprom_is_ready
    \ingroup avr_eeprom
    \returns 1 if EEPROM is ready for a new read/write operation, 0 if not.
 */
#if defined (__DOXYGEN__)
# define eeprom_is_ready()
#elif   defined (NVM_STATUS)
# define eeprom_is_ready()  bit_is_clear (NVM_STATUS, NVM_NVMBUSY_bp)
#elif   defined (NVMCTRL_STATUS)
# define eeprom_is_ready()  bit_is_clear (NVMCTRL_STATUS, NVMCTRL_EEBUSY_bp)
#elif   defined (DEECR)
# define eeprom_is_ready()  bit_is_clear (DEECR, BSY)
#elif   defined (EEPE)
# define eeprom_is_ready()  bit_is_clear (EECR, EEPE)
#else
# define eeprom_is_ready()  bit_is_clear (EECR, EEWE)
#endif


/** \def eeprom_busy_wait
    \ingroup avr_eeprom
    Loops until the eeprom is no longer busy.
    \returns Nothing.
 */
#define eeprom_busy_wait() do {} while (!eeprom_is_ready())


/** \ingroup avr_eeprom
    Read one byte from EEPROM address \a __p.
 */
uint8_t eeprom_read_byte (const uint8_t *__p) __ATTR_PURE__;

/** \ingroup avr_eeprom
    Read one 16-bit word (little endian) from EEPROM address \a __p.
 */
uint16_t eeprom_read_word (const uint16_t *__p) __ATTR_PURE__;

/** \ingroup avr_eeprom
    Read one 32-bit double word (little endian) from EEPROM address \a __p.
 */
uint32_t eeprom_read_dword (const uint32_t *__p) __ATTR_PURE__;

/** \ingroup avr_eeprom
    Read one 64-bit quad word (little endian) from EEPROM address \a __p.
 */
#if defined(__DOXYGEN__) || __SIZEOF_LONG_LONG__ == 8
uint64_t eeprom_read_qword (const uint64_t *__p) __ATTR_PURE__;
#endif

/** \ingroup avr_eeprom
    Read one float value (little endian) from EEPROM address \a __p.
 */
float eeprom_read_float (const float *__p) __ATTR_PURE__;

/** \ingroup avr_eeprom
    Read one double value (little endian) from EEPROM address \a __p.
 */
#if defined(__DOXYGEN__)
double eeprom_read_double (const double *__p);
#elif __SIZEOF_DOUBLE__ == 4
double eeprom_read_double (const double *__p) __asm("eeprom_read_dword");
#elif __SIZEOF_DOUBLE__ == 8
double eeprom_read_double (const double *__p) __asm("eeprom_read_qword");
#endif

/** \ingroup avr_eeprom
    Read one long double value (little endian) from EEPROM address \a __p.
 */
#if defined(__DOXYGEN__)
long double eeprom_read_long_double (const long double *__p);
#elif __SIZEOF_LONG_DOUBLE__ == 4
long double eeprom_read_long_double (const long double *__p) __asm("eeprom_read_dword");
#elif __SIZEOF_LONG_DOUBLE__ == 8
long double eeprom_read_long_double (const long double *__p) __asm("eeprom_read_qword");
#endif

/** \ingroup avr_eeprom
    Read a block of \a __n bytes from EEPROM address \a __src to SRAM
    \a __dst.
 */
void eeprom_read_block (void *__dst, const void *__src, size_t __n);
```

```
/** \ingroup avr_eeprom
    Write a byte \a __value to EEPROM address \a __p.
 */
void eeprom_write_byte (uint8_t *__p, uint8_t __value);

/** \ingroup avr_eeprom
    Write a word \a __value to EEPROM address \a __p.
 */
void eeprom_write_word (uint16_t *__p, uint16_t __value);

/** \ingroup avr_eeprom
    Write a 32-bit double word \a __value to EEPROM address \a __p.
 */
void eeprom_write_dword (uint32_t *__p, uint32_t __value);

/** \ingroup avr_eeprom
    Write a 64-bit quad word \a __value to EEPROM address \a __p.
 */
#if defined(__DOXYGEN__) || __SIZEOF_LONG_LONG__ == 8
void eeprom_write_qword (uint64_t *__p, uint64_t __value);
#endif

/** \ingroup avr_eeprom
    Write a float \a __value to EEPROM address \a __p.
 */
void eeprom_write_float (float *__p, float __value);

/** \ingroup avr_eeprom
    Write a double \a __value to EEPROM address \a __p.
 */
#if defined(__DOXYGEN__)
void eeprom_write_double (double *__p, double __value);
#elif __SIZEOF_DOUBLE__ == 4
void eeprom_write_double (double *__p, double __value) __asm("eeprom_write_dword");
#elif __SIZEOF_DOUBLE__ == 8
void eeprom_write_double (double *__p, double __value) __asm("eeprom_write_qword");
#endif

/** \ingroup avr_eeprom
    Write a long double \a __value to EEPROM address \a __p.
 */
#if defined(__DOXYGEN__)
void eeprom_write_long_double (long double *__p, long double __value);
#elif __SIZEOF_LONG_DOUBLE__ == 4
void eeprom_write_long_double (long double *__p, long double __value) __asm("eeprom_write_dword");
#elif __SIZEOF_LONG_DOUBLE__ == 8
void eeprom_write_long_double (long double *__p, long double __value) __asm("eeprom_write_qword");
#endif

/** \ingroup avr_eeprom
    Write a block of \a __n bytes to EEPROM address \a __dst from \a __src.
    \note The argument order is mismatch with common functions like strcpy().
 */
void eeprom_write_block (const void *__src, void *__dst, size_t __n);


/** \ingroup avr_eeprom
    Update a byte \a __value at EEPROM address \a __p.
 */
void eeprom_update_byte (uint8_t *__p, uint8_t __value);

/** \ingroup avr_eeprom
    Update a word \a __value at EEPROM address \a __p.
 */
void eeprom_update_word (uint16_t *__p, uint16_t __value);

/** \ingroup avr_eeprom
    Update a 32-bit double word \a __value at EEPROM address \a __p.
 */
void eeprom_update_dword (uint32_t *__p, uint32_t __value);

/** \ingroup avr_eeprom
    Update a 64-bit quad word \a __value at EEPROM address \a __p.
 */
#if defined(__DOXYGEN__) || __SIZEOF_LONG_LONG__ == 8
void eeprom_update_qword (uint64_t *__p, uint64_t __value);
#endif

/** \ingroup avr_eeprom
    Update a float \a __value at EEPROM address \a __p.
 */
void eeprom_update_float (float *__p, float __value);

/** \ingroup avr_eeprom
    Update a double \a __value at EEPROM address \a __p.
 */
#if defined(__DOXYGEN__)
```

```
void eeprom_update_double (double *__p, double __value);
#elif __SIZEOF_DOUBLE__ == 4
void eeprom_update_double (double *__p, double __value) __asm("eeprom_update_dword");
#elif __SIZEOF_DOUBLE__ == 8
void eeprom_update_double (double *__p, double __value) __asm("eeprom_update_qword");
#endif

/** \ingroup avr_eeprom
    Update a long double \a __value at EEPROM address \a __p.
 */
#if defined(__DOXYGEN__)
void eeprom_update_long_double (long double *__p, long double __value);
#elif __SIZEOF_LONG_DOUBLE__ == 4
void eeprom_update_long_double (long double *__p, long double __value) __asm("eeprom_update_dword");
#elif __SIZEOF_LONG_DOUBLE__ == 8
void eeprom_update_long_double (long double *__p, long double __value) __asm("eeprom_update_qword");
#endif

/** \ingroup avr_eeprom
    Update a block of \a __n bytes at EEPROM address \a __dst from \a __src.
    \note The argument order is mismatch with common functions like strcpy().
 */
void eeprom_update_block (const void *__src, void *__dst, size_t __n);


/** \name IAR C compatibility defines   */
/**@{*/

/** \def _EEPUT
    \ingroup avr_eeprom
    Write a byte to EEPROM. Compatibility define for IAR C. */
#define _EEPUT(addr, val) eeprom_write_byte ((uint8_t *)(addr), (uint8_t)(val))

/** \def __EEPUT
    \ingroup avr_eeprom
    Write a byte to EEPROM. Compatibility define for IAR C. */
#define __EEPUT(addr, val) eeprom_write_byte ((uint8_t *)(addr), (uint8_t)(val))

/** \def _EEGET
    \ingroup avr_eeprom
    Read a byte from EEPROM. Compatibility define for IAR C.    */
#define _EEGET(var, addr) (var) = eeprom_read_byte ((const uint8_t *)(addr))

/** \def __EEGET
    \ingroup avr_eeprom
    Read a byte from EEPROM. Compatibility define for IAR C.    */
#define __EEGET(var, addr) (var) = eeprom_read_byte ((const uint8_t *)(addr))

/**@}*/

#ifdef __cplusplus
}
#endif

#endif  /* !__ASSEMBLER__ */
#endif  /* E2END || defined(__DOXYGEN__) || defined(__COMPILING_AVR_LIBC__) */
#endif  /* !_AVR_EEPROM_H_ */
```

23.17 fuse.h File Reference

23.18 fuse.h

Go to the documentation of this file.

```
/* Copyright (c) 2007, Atmel Corporation
   All rights reserved.

   Redistribution and use in source and binary forms, with or without
   modification, are permitted provided that the following conditions are met:

   * Redistributions of source code must retain the above copyright
     notice, this list of conditions and the following disclaimer.

   * Redistributions in binary form must reproduce the above copyright
     notice, this list of conditions and the following disclaimer in
     the documentation and/or other materials provided with the
     distribution.

   * Neither the name of the copyright holders nor the names of
     contributors may be used to endorse or promote products derived
     from this software without specific prior written permission.
```

```
  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
  AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
  IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
  ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
  LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
  CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
  SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
  INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
  CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
  ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
  POSSIBILITY OF SUCH DAMAGE. */

/* $Id$ */

/* avr/fuse.h - Fuse API */

#ifndef _AVR_FUSE_H_
#define _AVR_FUSE_H_ 1

/* This file must be explicitly included by <avr/io.h>. */
#if !defined(_AVR_IO_H_)
#error "You must #include <avr/io.h> and not <avr/fuse.h> by itself."
#endif


/** \file */
/** \defgroup avr_fuse <avr/fuse.h>: Fuse Support

    \par Introduction

    The Fuse API allows a user to specify the fuse settings for the specific
    AVR device they are compiling for. These fuse settings will be placed
    in a special section in the ELF output file, after linking.

    Programming tools can take advantage of the fuse information embedded in
    the ELF file, by extracting this information and determining if the fuses
    need to be programmed before programming the Flash and EEPROM memories.
    This also allows a single ELF file to contain all the
    information needed to program an AVR.

    To use the Fuse API, include the <avr/io.h> header file, which in turn
    automatically includes the individual I/O header file and the <avr/fuse.h>
    file. These other two files provides everything necessary to set the AVR
    fuses.

    \par Fuse API

    Each I/O header file must define the FUSE_MEMORY_SIZE macro which is
    defined to the number of fuse bytes that exist in the AVR device.

    A new type, __fuse_t, is defined as a structure. The number of fields in
    this structure are determined by the number of fuse bytes in the
    FUSE_MEMORY_SIZE macro.

    If FUSE_MEMORY_SIZE == 1, there is only a single field: byte, of type
    unsigned char.

    If FUSE_MEMORY_SIZE == 2, there are two fields: low, and high, of type
    unsigned char.

    If FUSE_MEMORY_SIZE == 3, there are three fields: low, high, and extended,
    of type unsigned char.

    If FUSE_MEMORY_SIZE > 3, there is a single field: byte, which is an array
    of unsigned char with the size of the array being FUSE_MEMORY_SIZE.

    A convenience macro, FUSEMEM, is defined as a GCC attribute for a
    custom-named section of ".fuse".

    A convenience macro, FUSES, is defined that declares a variable, __fuse, of
    type __fuse_t with the attribute defined by FUSEMEM. This variable
    allows the end user to easily set the fuse data.

    \note If a device-specific I/O header file has previously defined FUSEMEM,
    then FUSEMEM is not redefined. If a device-specific I/O header file has
    previously defined FUSES, then FUSES is not redefined.

    Each AVR device I/O header file has a set of defined macros which specify the
    actual fuse bits available on that device. The AVR fuses have inverted
    values, logical 1 for an unprogrammed (disabled) bit and logical 0 for a
    programmed (enabled) bit. The defined macros for each individual fuse
    bit represent this in their definition by a bit-wise inversion of a mask.
    For example, the FUSE_EESAVE fuse in the ATmega128 is defined as:
    \code
    #define FUSE_EESAVE      ~_BV(3)
    \endcode
```

```
\note The _BV macro creates a bit mask from a bit number. It is then
inverted to represent logical values for a fuse memory byte.

To combine the fuse bits macros together to represent a whole fuse byte,
use the bitwise AND operator, like so:
\code
(FUSE_BOOTSZ0 & FUSE_BOOTSZ1 & FUSE_EESAVE & FUSE_SPIEN & FUSE_JTAGEN)
\endcode

Each device I/O header file also defines macros that provide default values
for each fuse byte that is available. LFUSE_DEFAULT is defined for a Low
Fuse byte. HFUSE_DEFAULT is defined for a High Fuse byte. EFUSE_DEFAULT
is defined for an Extended Fuse byte.

If FUSE_MEMORY_SIZE > 3, then the I/O header file defines macros that
provide default values for each fuse byte like so:
FUSE0_DEFAULT
FUSE1_DEFAULT
FUSE2_DEFAULT
FUSE3_DEFAULT
FUSE4_DEFAULT
....

\par API Usage Example

Putting all of this together is easy. Using C99's designated initializers:

\code
#include <avr/io.h>

FUSES =
{
    .low = LFUSE_DEFAULT,
    .high = (FUSE_BOOTSZ0 & FUSE_BOOTSZ1 & FUSE_EESAVE & FUSE_SPIEN & FUSE_JTAGEN),
    .extended = EFUSE_DEFAULT,
};

int main(void)
{
    return 0;
}
\endcode

Or, using the variable directly instead of the FUSES macro,

\code
#include <avr/io.h>

__fuse_t __fuse __attribute__((section (".fuse"))) =
{
    .low = LFUSE_DEFAULT,
    .high = (FUSE_BOOTSZ0 & FUSE_BOOTSZ1 & FUSE_EESAVE & FUSE_SPIEN & FUSE_JTAGEN),
    .extended = EFUSE_DEFAULT,
};

int main(void)
{
    return 0;
}
\endcode

If you are compiling in C++, you cannot use the designated intializers so
you must do:

\code
#include <avr/io.h>

FUSES =
{
    LFUSE_DEFAULT, // .low
    (FUSE_BOOTSZ0 & FUSE_BOOTSZ1 & FUSE_EESAVE & FUSE_SPIEN & FUSE_JTAGEN), // .high
    EFUSE_DEFAULT, // .extended
};

int main(void)
{
    return 0;
}
\endcode


However there are a number of caveats that you need to be aware of to
use this API properly.

Be sure to include <avr/io.h> to get all of the definitions for the API.
The FUSES macro defines a global variable to store the fuse data. This
variable is assigned to its own linker section. Assign the desired fuse
```

```
    values immediately in the variable initialization.

    The .fuse section in the ELF file will get its values from the initial
    variable assignment ONLY. This means that you can NOT assign values to
    this variable in functions and the new values will not be put into the
    ELF .fuse section.

    The global variable is declared in the FUSES macro has two leading
    underscores, which means that it is reserved for the "implementation",
    meaning the library, so it will not conflict with a user-named variable.

    You must initialize ALL fields in the __fuse_t structure. This is because
    the fuse bits in all bytes default to a logical 1, meaning unprogrammed.
    Normal uninitialized data defaults to all locgial zeros. So it is vital that
    all fuse bytes are initialized, even with default data. If they are not,
    then the fuse bits may not programmed to the desired settings.

    Be sure to have the -mmcu=<em>device</em> flag in your compile command line and
    your linker command line to have the correct device selected and to have
    the correct I/O header file included when you include <avr/io.h>.

    You can print out the contents of the .fuse section in the ELF file by
    using this command line:
    \code
    avr-objdump -s -j .fuse <ELF file>
    \endcode
    The section contents shows the address on the left, then the data going from
    lower address to a higher address, left to right.

*/

#if !(defined(__ASSEMBLER__) || defined(__DOXYGEN__))

#ifndef FUSEMEM
#define FUSEMEM  __attribute__((__used__, __section__ (".fuse")))
#endif

#if FUSE_MEMORY_SIZE > 3

typedef struct
{
    unsigned char byte[FUSE_MEMORY_SIZE];
} __fuse_t;


#elif FUSE_MEMORY_SIZE == 3

typedef struct
{
    unsigned char low;
    unsigned char high;
    unsigned char extended;
} __fuse_t;

#elif FUSE_MEMORY_SIZE == 2

typedef struct
{
    unsigned char low;
    unsigned char high;
} __fuse_t;

#elif FUSE_MEMORY_SIZE == 1

typedef struct
{
    unsigned char byte;
} __fuse_t;

#endif

#if !defined(FUSES)
  #if defined(__AVR_XMEGA__)
    #define FUSES NVM_FUSES_t __fuse FUSEMEM
  #else
    #define FUSES __fuse_t __fuse FUSEMEM
  #endif
#endif


#endif /* !(__ASSEMBLER__ || __DOXYGEN__) */

#endif /* _AVR_FUSE_H_ */
```

23.19 interrupt.h File Reference

Macros

Global manipulation of the interrupt flag

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

Handling interrupts frequently requires attention regarding atomic access to objects that could be altered by code running within an interrupt context, see <util/atomic.h>.

Frequently, interrupts are being disabled for periods of time in order to perform certain operations without being disturbed; see Problems with reordering code for things to be taken into account with respect to compiler optimizations.

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

Macros for writing interrupt handler functions

- #define ISR(vector, attributes)
- #define SIGNAL(vector)
- #define EMPTY_INTERRUPT(vector)
- #define ISR_ALIAS(vector, target_vector)
- #define reti() __asm__ __volatile__ ("reti" ::: "memory")
- #define BADISR_vect

ISR attributes

- #define ISR_BLOCK
- #define ISR_NOBLOCK
- #define ISR_NAKED
- #define ISR_FLATTEN
- #define ISR_NOICF
- #define ISR_NOGCCISR
- #define ISR_ALIASOF(target_vector)

23.20 interrupt.h

Go to the documentation of this file.

```
/* Copyright (c) 2002,2005,2007 Marek Michalkiewicz
   Copyright (c) 2007, Dean Camera

   All rights reserved.

   Redistribution and use in source and binary forms, with or without
   modification, are permitted provided that the following conditions are met:

   * Redistributions of source code must retain the above copyright
     notice, this list of conditions and the following disclaimer.

   * Redistributions in binary form must reproduce the above copyright
     notice, this list of conditions and the following disclaimer in
     the documentation and/or other materials provided with the
     distribution.

   * Neither the name of the copyright holders nor the names of
     contributors may be used to endorse or promote products derived
     from this software without specific prior written permission.

  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
  AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
  IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
  ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
  LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
  CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
  SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
  INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
```

```
  CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
  ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
  POSSIBILITY OF SUCH DAMAGE. */

/* $Id$ */

#ifndef _AVR_INTERRUPT_H_
#define _AVR_INTERRUPT_H_

#include <avr/io.h>

#if !defined(__DOXYGEN__) && !defined(__STRINGIFY)
/* Auxiliary macro for ISR_ALIAS(). */
#define __STRINGIFY(x) #x
#endif /* !defined(__DOXYGEN__) */

/** \file */
/**@{*/


/** \name Global manipulation of the interrupt flag

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

    Handling interrupts frequently requires attention regarding atomic
    access to objects that could be altered by code running within an
    interrupt context, see <util/atomic.h>.

    Frequently, interrupts are being disabled for periods of time in
    order to perform certain operations without being disturbed; see
    \ref optim_code_reorder for things to be taken into account with
    respect to compiler optimizations.
*/

/** \def sei()
    \ingroup avr_interrupts

    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.  However, the macro also implies a <i>memory barrier</i>
    which can cause additional loss of optimization.

    In order to implement atomic access to multi-byte objects,
    consider using the macros from <util/atomic.h>, rather than
    implementing them manually with cli() and sei().
*/
# define sei()  __asm__ __volatile__ ("sei" ::: "memory")

/** \def cli()
    \ingroup avr_interrupts

    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.  However, the macro also implies a <i>memory barrier</i>
    which can cause additional loss of optimization.

    In order to implement atomic access to multi-byte objects,
    consider using the macros from <util/atomic.h>, rather than
    implementing them manually with cli() and sei().
*/
# define cli()  __asm__ __volatile__ ("cli" ::: "memory")


/** \name Macros for writing interrupt handler functions */


#if defined(__DOXYGEN__)
/** \def ISR(vector [, attributes])
    \ingroup avr_interrupts

    Introduces an interrupt handler function (interrupt service
    routine) that runs with global interrupts initially disabled
    by default with no attributes specified.

    The attributes are optional and alter the behaviour and resultant
    generated code of the interrupt routine. Multiple attributes may
    be used for a single function, with a space seperating each
    attribute.

    Valid attributes are #ISR_BLOCK, #ISR_NOBLOCK, #ISR_NAKED,
    #ISR_FLATTEN, #ISR_NOICF, #ISR_NOGCCISR and ISR_ALIASOF(vect).

    \c vector must be one of the interrupt vector names that are
    valid for the particular MCU type.
*/
#  define ISR(vector, [attributes])
```

```
#else  /* real code */

#if (__GNUC__ == 4 && __GNUC_MINOR__ >= 1) || (__GNUC__ > 4)
#  define __INTR_ATTRS __used__, __externally_visible__
#else /* GCC < 4.1 */
#  define __INTR_ATTRS __used__
#endif

#ifdef __cplusplus
#  define ISR(vector, ...)            \
    extern "C" void vector (void) __attribute__ ((__signal__,__INTR_ATTRS)) __VA_ARGS__; \
    void vector (void)
#else
#  define ISR(vector, ...)            \
    void vector (void) __attribute__ ((__signal__,__INTR_ATTRS)) __VA_ARGS__; \
    void vector (void)
#endif

#endif /* DOXYGEN */

#if defined(__DOXYGEN__)
/** \def SIGNAL(vector)
    \ingroup avr_interrupts

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

    This is the same as the ISR macro without optional attributes.
    \deprecated Do not use SIGNAL() in new code. Use ISR() instead.
*/
#  define SIGNAL(vector)
#else  /* real code */

#ifdef __cplusplus
#  define SIGNAL(vector)                    \
    extern "C" void vector(void) __attribute__ ((__signal__, __INTR_ATTRS));    \
    void vector (void)
#else
#  define SIGNAL(vector)                    \
    void vector (void) __attribute__ ((__signal__, __INTR_ATTRS));      \
    void vector (void)
#endif

#endif /* DOXYGEN */

#if defined(__DOXYGEN__)
/** \def EMPTY_INTERRUPT(vector)
    \ingroup avr_interrupts

    Defines an empty interrupt handler function. This will not generate
    any prolog or epilog code and will only return from the #ISR. Do not
    define a function body as this will define it for you.
    Example:
    \code EMPTY_INTERRUPT(ADC_vect);\endcode */
#  define EMPTY_INTERRUPT(vector)
#else  /* real code */

#ifdef __cplusplus
#  define EMPTY_INTERRUPT(vector)                \
    extern "C" void vector(void) __attribute__ ((__signal__,__naked__,__INTR_ATTRS));    \
    void vector (void) {  __asm__ __volatile__ ("reti" ::: "memory"); }
#else
#  define EMPTY_INTERRUPT(vector)                \
    void vector (void) __attribute__ ((__signal__,__naked__,__INTR_ATTRS));    \
    void vector (void) { __asm__ __volatile__ ("reti" ::: "memory"); }
#endif

#endif /* DOXYGEN */

#if defined(__DOXYGEN__)
/** \def ISR_ALIAS(vector, target_vector)
    \ingroup avr_interrupts

    Aliases a given vector to another one in the same manner as the
    ISR_ALIASOF attribute for the ISR() macro. Unlike the ISR_ALIASOF
    attribute macro however, this is compatible for all versions of
    GCC rather than just GCC version 4.2 onwards.

    \note This macro creates a trampoline function for the aliased
    macro.  This will result in a two cycle penalty for the aliased
    vector compared to the ISR the vector is aliased to, due to the
    JMP/RJMP opcode used.

    \deprecated
    For new code, the use of ISR(..., ISR_ALIASOF(...))  is
    recommended.

```

```
    Example:
    \code
    ISR(INT0_vect)
    {
        PORTB = 42;
    }

    ISR_ALIAS(INT1_vect, INT0_vect);
    \endcode

*/
#  define ISR_ALIAS(vector, target_vector)
#else /* real code */

#ifdef __cplusplus
#    define ISR_ALIAS(vector, tgt) extern "C" void vector (void) \
    __attribute__((__signal__, __naked__, __INTR_ATTRS)); \
    void vector (void) { __asm__ __volatile__ ("%~jmp " __STRINGIFY(tgt) ::); }
#else     /* !__cplusplus */
#  define ISR_ALIAS(vector, tgt) void vector (void) \
    __attribute__((__signal__, __naked__, __INTR_ATTRS)); \
    void vector (void) { __asm__ __volatile__ ("%~jmp " __STRINGIFY(tgt) ::); }
#endif  /* __cplusplus */

#endif /* DOXYGEN */

/** \def reti()
    \ingroup avr_interrupts

    Returns from an interrupt routine, enabling global interrupts. This should
    be the last command executed before leaving an #ISR defined with the
    #ISR_NAKED attribute.

    This macro actually compiles into a single line of assembly, so there is
    no function call overhead.

    \note According to the GCC documentation, the only code supported in
    naked functions is \ref inline_asm "inline assembly".
*/
#  define reti()  __asm__ __volatile__ ("reti" ::: "memory")

#if defined(__DOXYGEN__)
/** \def BADISR_vect
    \ingroup avr_interrupts

    \code #include <avr/interrupt.h> \endcode

    This is a vector which is aliased to __vector_default, the vector
    executed when an IRQ fires with no accompanying ISR handler. This
    may be used along with the ISR() macro to create a catch-all for
    undefined but used ISRs for debugging purposes.
*/
#  define BADISR_vect
#else  /* !DOXYGEN */
#  define BADISR_vect __vector_default
#endif /* DOXYGEN */

/** \name ISR attributes */

#if defined(__DOXYGEN__)
/** \def ISR_BLOCK
    \ingroup avr_interrupts

    Identical to an ISR with no attributes specified. Global
    interrupts are initially disabled by the AVR hardware when
    entering the ISR, without the compiler modifying this state.

    Use this attribute in the attributes parameter of the #ISR macro.
*/
#  define ISR_BLOCK

/** \def ISR_NOBLOCK
    \ingroup avr_interrupts

    ISR runs with global interrupts initially enabled.  The interrupt
    enable flag is activated by the compiler as early as possible
    within the ISR to ensure minimal processing delay for nested
    interrupts.

    This may be used to create nested ISRs, however care should be
    taken to avoid stack overflows, or to avoid infinitely entering
    the ISR for those cases where the AVR hardware does not clear the
    respective interrupt flag before entering the ISR.

    Use this attribute in the attributes parameter of the #ISR macro.
*/
#  define ISR_NOBLOCK
```

```
/** \def ISR_NAKED
    \ingroup avr_interrupts

    ISR is created with no prologue or epilogue code. The user code is
    responsible for preservation of the machine state including the
    SREG register, as well as placing a reti() at the end of the
    interrupt routine.

    Use this attribute in the attributes parameter of the #ISR macro.

    \note According to GCC documentation, the only code supported in
    naked functions is \ref inline_asm "inline assembly".
*/
#  define ISR_NAKED

/** \def ISR_FLATTEN
    \ingroup avr_interrupts

    The compiler will try to inline all called function into the ISR.
    This has an effect with GCC 4.6 and newer only.

    Use this attribute in the attributes parameter of the #ISR macro.
*/
#  define ISR_FLATTEN

/** \def ISR_NOICF
    \ingroup avr_interrupts

    Avoid identical-code-folding optimization against this ISR.
    This has an effect with GCC 5 and newer only.

    Use this attribute in the attributes parameter of the #ISR macro.
*/
#  define ISR_NOICF

/** \def ISR_NOGCCISR
    \ingroup avr_interrupts

    Do not generate
    <a href="https://sourceware.org/binutils/docs/as/AVR-Pseudo-Instructions.html">\c __gcc_isr pseudo instructions</a>
    for this ISR.
    This has an effect with
    <a href="https://gcc.gnu.org/gcc-8/changes.html#avr">GCC 8</a>
    and newer only.

    Use this attribute in the attributes parameter of the #ISR macro.
*/
#  define ISR_NOGCCISR

/** \def ISR_ALIASOF(target_vector)
    \ingroup avr_interrupts

    The ISR is linked to another ISR, specified by the vect parameter.
    This is compatible with GCC 4.2 and greater only.

    Use this attribute in the attributes parameter of the #ISR macro.
    Example:
    \code
    ISR (INT0_vect)
    {
        PORTB = 42;
    }

    ISR (INT1_vect, ISR_ALIASOF (INT0_vect));
    \endcode
*/
#  define ISR_ALIASOF(target_vector)
#else  /* !DOXYGEN */
#  define ISR_BLOCK /* empty */
/* FIXME: This won't work with older versions of avr-gcc as ISR_NOBLOCK
          will use 'signal' and 'interrupt' at the same time.  */
#  define ISR_NOBLOCK    __attribute__((__interrupt__))
#  define ISR_NAKED      __attribute__((__naked__))

#if (__GNUC__ == 4 && __GNUC_MINOR__ >= 6) || (__GNUC__ >= 5)
#  define ISR_FLATTEN    __attribute__((__flatten__))
#else
#  define ISR_FLATTEN    /* empty */
#endif /* has flatten (GCC 4.6+) */

#if defined (__has_attribute)
#if __has_attribute (__no_icf__)
#  define ISR_NOICF      __attribute__((__no_icf__))
#else
#  define ISR_NOICF      /* empty */
```

```
#endif /* has no_icf */

#if __has_attribute (__no_gccisr__)
#  define ISR_NOGCCISR      __attribute__((__no_gccisr__))
#else
#  define ISR_NOGCCISR      /* empty */
#endif /* has no_gccisr */
#endif /* has __has_attribute (GCC 5+) */

#  define ISR_ALIASOF(v) __attribute__((__alias__(__STRINGIFY(v))))
#endif /* DOXYGEN */

/**@}*/

#endif
```

23.21 io.h File Reference

23.22 io.h

Go to the documentation of this file.

```
/* Copyright (c) 2002,2003,2005,2006,2007 Marek Michalkiewicz, Joerg Wunsch
   Copyright (c) 2007 Eric B. Weddington
   All rights reserved.

   Redistribution and use in source and binary forms, with or without
   modification, are permitted provided that the following conditions are met:

   * Redistributions of source code must retain the above copyright
     notice, this list of conditions and the following disclaimer.

   * Redistributions in binary form must reproduce the above copyright
     notice, this list of conditions and the following disclaimer in
     the documentation and/or other materials provided with the
     distribution.

   * Neither the name of the copyright holders nor the names of
     contributors may be used to endorse or promote products derived
     from this software without specific prior written permission.

  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
  AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
  IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
  ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
  LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
  CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
  SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
  INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
  CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
  ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
  POSSIBILITY OF SUCH DAMAGE. */

/* $Id$ */

/** \file */
/** \defgroup avr_io <avr/io.h>: AVR device-specific IO definitions
    \code #include <avr/io.h> \endcode

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

    Note that this file always includes the following files:
    \code
    #include <avr/sfr_defs.h>
    #include <avr/portpins.h>
    #include <avr/common.h>
    #include <avr/version.h>
    \endcode
    See \ref avr_sfr for more details about that header file.

    Included are definitions of the IO register set and their
    respective bit values as specified in the Atmel documentation.
    Note that inconsistencies in naming conventions,
    so even identical functions sometimes get different names on
```

```
    different devices.

    Also included are the specific names useable for interrupt
    function definitions as documented
    \ref avr_signames "here".

    Finally, the following macros are defined:

    - \b RAMEND
    <br>
    The last on-chip RAM address.
    <br>
    - \b XRAMEND
    <br>
    The last possible RAM location that is addressable. This is equal to
    RAMEND for devices that do not allow for external RAM. For devices
    that allow external RAM, this will be larger than RAMEND.
    <br>
    - \b E2END
    <br>
    The last EEPROM address.
    <br>
    - \b FLASHEND
    <br>
    The last byte address in the Flash program space.
    <br>
    - \b SPM_PAGESIZE
    <br>
    For devices with bootloader support, the flash pagesize
    (in bytes) to be used for the \c SPM instruction.
    - \b E2PAGESIZE
    <br>
    The size of the EEPROM page.

*/

#ifndef _AVR_IO_H_
#define _AVR_IO_H_

#include <avr/sfr_defs.h>

#if defined (__AVR_AT94K__)
#  include <avr/ioat94k.h>
#elif defined (__AVR_AT43USB320__)
#  include <avr/io43u32x.h>
#elif defined (__AVR_AT43USB355__)
#  include <avr/io43u35x.h>
#elif defined (__AVR_AT76C711__)
#  include <avr/io76c711.h>
#elif defined (__AVR_AT86RF401__)
#  include <avr/io86r401.h>
#elif defined (__AVR_AT90PWM1__)
#  include <avr/io90pwm1.h>
#elif defined (__AVR_AT90PWM2__)
#  include <avr/io90pwmx.h>
#elif defined (__AVR_AT90PWM2B__)
#  include <avr/io90pwm2b.h>
#elif defined (__AVR_AT90PWM3__)
#  include <avr/io90pwmx.h>
#elif defined (__AVR_AT90PWM3B__)
#  include <avr/io90pwm3b.h>
#elif defined (__AVR_AT90PWM216__)
#  include <avr/io90pwm216.h>
#elif defined (__AVR_AT90PWM316__)
#  include <avr/io90pwm316.h>
#elif defined (__AVR_AT90PWM161__)
#  include <avr/io90pwm161.h>
#elif defined (__AVR_AT90PWM81__)
#  include <avr/io90pwm81.h>
#elif defined (__AVR_ATmega8U2__)
#  include <avr/iom8u2.h>
#elif defined (__AVR_ATmega16M1__)
#  include <avr/iom16m1.h>
#elif defined (__AVR_ATmega16U2__)
#  include <avr/iom16u2.h>
#elif defined (__AVR_ATmega16U4__)
#  include <avr/iom16u4.h>
#elif defined (__AVR_ATmega32C1__)
#  include <avr/iom32c1.h>
#elif defined (__AVR_ATmega32M1__)
#  include <avr/iom32m1.h>
#elif defined (__AVR_ATmega32U2__)
#  include <avr/iom32u2.h>
#elif defined (__AVR_ATmega32U4__)
#  include <avr/iom32u4.h>
#elif defined (__AVR_ATmega32U6__)
#  include <avr/iom32u6.h>
```

```
#elif defined (__AVR_ATmega64C1__)
#  include <avr/iom64c1.h>
#elif defined (__AVR_ATmega64M1__)
#  include <avr/iom64m1.h>
#elif defined (__AVR_ATmega128__)
#  include <avr/iom128.h>
#elif defined (__AVR_ATmega128A__)
#  include <avr/iom128a.h>
#elif defined (__AVR_ATmega1280__)
#  include <avr/iom1280.h>
#elif defined (__AVR_ATmega1281__)
#  include <avr/iom1281.h>
#elif defined (__AVR_ATmega1284__)
#  include <avr/iom1284.h>
#elif defined (__AVR_ATmega1284P__)
#  include <avr/iom1284p.h>
#elif defined (__AVR_ATmega128RFA1__)
#  include <avr/iom128rfa1.h>
#elif defined (__AVR_ATmega1284RFR2__)
#  include <avr/iom1284rfr2.h>
#elif defined (__AVR_ATmega128RFR2__)
#  include <avr/iom128rfr2.h>
#elif defined (__AVR_ATmega2564RFR2__)
#  include <avr/iom2564rfr2.h>
#elif defined (__AVR_ATmega256RFR2__)
#  include <avr/iom256rfr2.h>
#elif defined (__AVR_ATmega2560__)
#  include <avr/iom2560.h>
#elif defined (__AVR_ATmega2561__)
#  include <avr/iom2561.h>
#elif defined (__AVR_AT90CAN32__)
#  include <avr/iocan32.h>
#elif defined (__AVR_AT90CAN64__)
#  include <avr/iocan64.h>
#elif defined (__AVR_AT90CAN128__)
#  include <avr/iocan128.h>
#elif defined (__AVR_AT90USB82__)
#  include <avr/iousb82.h>
#elif defined (__AVR_AT90USB162__)
#  include <avr/iousb162.h>
#elif defined (__AVR_AT90USB646__)
#  include <avr/iousb646.h>
#elif defined (__AVR_AT90USB647__)
#  include <avr/iousb647.h>
#elif defined (__AVR_AT90USB1286__)
#  include <avr/iousb1286.h>
#elif defined (__AVR_AT90USB1287__)
#  include <avr/iousb1287.h>
#elif defined (__AVR_ATmega644RFR2__)
#  include <avr/iom644rfr2.h>
#elif defined (__AVR_ATmega64RFR2__)
#  include <avr/iom64rfr2.h>
#elif defined (__AVR_ATmega64__)
#  include <avr/iom64.h>
#elif defined (__AVR_ATmega64A__)
#  include <avr/iom64a.h>
#elif defined (__AVR_ATmega640__)
#  include <avr/iom640.h>
#elif defined (__AVR_ATmega644__)
#  include <avr/iom644.h>
#elif defined (__AVR_ATmega644A__)
#  include <avr/iom644a.h>
#elif defined (__AVR_ATmega644P__)
#  include <avr/iom644p.h>
#elif defined (__AVR_ATmega644PA__)
#  include <avr/iom644pa.h>
#elif defined (__AVR_ATmega645__) || defined (__AVR_ATmega645A__) || defined (__AVR_ATmega645P__)
#  include <avr/iom645.h>
#elif defined (__AVR_ATmega6450__) || defined (__AVR_ATmega6450A__) || defined (__AVR_ATmega6450P__)
#  include <avr/iom6450.h>
#elif defined (__AVR_ATmega649__) || defined (__AVR_ATmega649A__)
#  include <avr/iom649.h>
#elif defined (__AVR_ATmega6490__) || defined (__AVR_ATmega6490A__) || defined (__AVR_ATmega6490P__)
#  include <avr/iom6490.h>
#elif defined (__AVR_ATmega649P__)
#  include <avr/iom649p.h>
#elif defined (__AVR_ATmega64HVE__)
#  include <avr/iom64hve.h>
#elif defined (__AVR_ATmega64HVE2__)
#  include <avr/iom64hve2.h>
#elif defined (__AVR_ATmega103__)
#  include <avr/iom103.h>
#elif defined (__AVR_ATmega32__)
#  include <avr/iom32.h>
#elif defined (__AVR_ATmega32A__)
#  include <avr/iom32a.h>
#elif defined (__AVR_ATmega323__)
```

```
#  include <avr/iom323.h>
#elif defined (__AVR_ATmega324P__) || defined (__AVR_ATmega324A__)
#  include <avr/iom324.h>
#elif defined (__AVR_ATmega324PA__)
#  include <avr/iom324pa.h>
#elif defined (__AVR_ATmega324PB__)
#  include <avr/iom324pb.h>
#elif defined (__AVR_ATmega325__) || defined (__AVR_ATmega325A__)
#  include <avr/iom325.h>
#elif defined (__AVR_ATmega325P__)
#  include <avr/iom325.h>
#elif defined (__AVR_ATmega325PA__)
#  include <avr/iom325pa.h>
#elif defined (__AVR_ATmega3250__) || defined (__AVR_ATmega3250A__)
#  include <avr/iom3250.h>
#elif defined (__AVR_ATmega3250P__)
#  include <avr/iom3250.h>
#elif defined (__AVR_ATmega3250PA__)
#  include <avr/iom3250pa.h>
#elif defined (__AVR_ATmega328P__) || defined (__AVR_ATmega328__)
#  include <avr/iom328p.h>
#elif defined (__AVR_ATmega328PB__)
#  include <avr/iom328pb.h>
#elif defined (__AVR_ATmega329__) || defined (__AVR_ATmega329A__)
#  include <avr/iom329.h>
#elif defined (__AVR_ATmega329P__) || defined (__AVR_ATmega329PA__)
#  include <avr/iom329.h>
#elif defined (__AVR_ATmega3290__) || defined (__AVR_ATmega3290A__)
#  include <avr/iom3290.h>
#elif defined (__AVR_ATmega3290P__)
#  include <avr/iom3290.h>
#elif defined (__AVR_ATmega3290PA__)
#  include <avr/iom3290pa.h>
#elif defined (__AVR_ATmega32HVB__)
#  include <avr/iom32hvb.h>
#elif defined (__AVR_ATmega32HVBREVB__)
#  include <avr/iom32hvbrevb.h>
#elif defined (__AVR_ATmega406__)
#  include <avr/iom406.h>
#elif defined (__AVR_ATmega16__)
#  include <avr/iom16.h>
#elif defined (__AVR_ATmega16A__)
#  include <avr/iom16a.h>
#elif defined (__AVR_ATmega161__)
#  include <avr/iom161.h>
#elif defined (__AVR_ATmega162__)
#  include <avr/iom162.h>
#elif defined (__AVR_ATmega163__)
#  include <avr/iom163.h>
#elif defined (__AVR_ATmega164P__) || defined (__AVR_ATmega164A__)
#  include <avr/iom164.h>
#elif defined (__AVR_ATmega164PA__)
#  include <avr/iom164pa.h>
#elif defined (__AVR_ATmega165__)
#  include <avr/iom165.h>
#elif defined (__AVR_ATmega165A__)
#  include <avr/iom165a.h>
#elif defined (__AVR_ATmega165P__)
#  include <avr/iom165p.h>
#elif defined (__AVR_ATmega165PA__)
#  include <avr/iom165pa.h>
#elif defined (__AVR_ATmega168__)
#  include <avr/iom168.h>
#elif defined (__AVR_ATmega168A__)
#  include <avr/iom168a.h>
#elif defined (__AVR_ATmega168P__)
#  include <avr/iom168p.h>
#elif defined (__AVR_ATmega168PA__)
#  include <avr/iom168pa.h>
#elif defined (__AVR_ATmega168PB__)
#  include <avr/iom168pb.h>
#elif defined (__AVR_ATmega169__) || defined (__AVR_ATmega169A__)
#  include <avr/iom169.h>
#elif defined (__AVR_ATmega169P__)
#  include <avr/iom169p.h>
#elif defined (__AVR_ATmega169PA__)
#  include <avr/iom169pa.h>
#elif defined (__AVR_ATmega8HVA__)
#  include <avr/iom8hva.h>
#elif defined (__AVR_ATmega16HVA__)
#  include <avr/iom16hva.h>
#elif defined (__AVR_ATmega16HVA2__)
#  include <avr/iom16hva2.h>
#elif defined (__AVR_ATmega16HVB__)
#  include <avr/iom16hvb.h>
#elif defined (__AVR_ATmega16HVBREVB__)
#  include <avr/iom16hvbrevb.h>
```

```
#elif defined (__AVR_ATmega8__)
#  include <avr/iom8.h>
#elif defined (__AVR_ATmega8A__)
#  include <avr/iom8a.h>
#elif defined (__AVR_ATmega48__)
#  include <avr/iom48.h>
#elif defined (__AVR_ATmega48A__)
#  include <avr/iom48a.h>
#elif defined (__AVR_ATmega48PA__)
#  include <avr/iom48pa.h>
#elif defined (__AVR_ATmega48PB__)
#  include <avr/iom48pb.h>
#elif defined (__AVR_ATmega48P__)
#  include <avr/iom48p.h>
#elif defined (__AVR_ATmega88__)
#  include <avr/iom88.h>
#elif defined (__AVR_ATmega88A__)
#  include <avr/iom88a.h>
#elif defined (__AVR_ATmega88P__)
#  include <avr/iom88p.h>
#elif defined (__AVR_ATmega88PA__)
#  include <avr/iom88pa.h>
#elif defined (__AVR_ATmega88PB__)
#  include <avr/iom88pb.h>
#elif defined (__AVR_ATmega8515__)
#  include <avr/iom8515.h>
#elif defined (__AVR_ATmega8535__)
#  include <avr/iom8535.h>
#elif defined (__AVR_AT90S8535__)
#  include <avr/io8535.h>
#elif defined (__AVR_AT90C8534__)
#  include <avr/io8534.h>
#elif defined (__AVR_AT90S8515__)
#  include <avr/io8515.h>
#elif defined (__AVR_AT90S4434__)
#  include <avr/io4434.h>
#elif defined (__AVR_AT90S4433__)
#  include <avr/io4433.h>
#elif defined (__AVR_AT90S4414__)
#  include <avr/io4414.h>
#elif defined (__AVR_ATtiny22__)
#  include <avr/iotn22.h>
#elif defined (__AVR_ATtiny26__)
#  include <avr/iotn26.h>
#elif defined (__AVR_AT90S2343__)
#  include <avr/io2343.h>
#elif defined (__AVR_AT90S2333__)
#  include <avr/io2333.h>
#elif defined (__AVR_AT90S2323__)
#  include <avr/io2323.h>
#elif defined (__AVR_AT90S2313__)
#  include <avr/io2313.h>
#elif defined (__AVR_ATtiny4__)
#  include <avr/iotn4.h>
#elif defined (__AVR_ATtiny5__)
#  include <avr/iotn5.h>
#elif defined (__AVR_ATtiny9__)
#  include <avr/iotn9.h>
#elif defined (__AVR_ATtiny10__)
#  include <avr/iotn10.h>
#elif defined (__AVR_ATtiny102__)
#  include <avr/iotn102.h>
#elif defined (__AVR_ATtiny104__)
#  include <avr/iotn104.h>
#elif defined (__AVR_ATtiny20__)
#  include <avr/iotn20.h>
#elif defined (__AVR_ATtiny40__)
#  include <avr/iotn40.h>
#elif defined (__AVR_ATtiny2313__)
#  include <avr/iotn2313.h>
#elif defined (__AVR_ATtiny2313A__)
#  include <avr/iotn2313a.h>
#elif defined (__AVR_ATtiny13__)
#  include <avr/iotn13.h>
#elif defined (__AVR_ATtiny13A__)
#  include <avr/iotn13a.h>
#elif defined (__AVR_ATtiny25__)
#  include <avr/iotn25.h>
#elif defined (__AVR_ATtiny4313__)
#  include <avr/iotn4313.h>
#elif defined (__AVR_ATtiny45__)
#  include <avr/iotn45.h>
#elif defined (__AVR_ATtiny85__)
#  include <avr/iotn85.h>
#elif defined (__AVR_ATtiny24__)
#  include <avr/iotn24.h>
#elif defined (__AVR_ATtiny24A__)
```

```
#  include <avr/iotn24a.h>
#elif defined (__AVR_ATtiny44__)
#  include <avr/iotn44.h>
#elif defined (__AVR_ATtiny44A__)
#  include <avr/iotn44a.h>
#elif defined (__AVR_ATtiny441__)
#  include <avr/iotn441.h>
#elif defined (__AVR_ATtiny84__)
#  include <avr/iotn84.h>
#elif defined (__AVR_ATtiny84A__)
#  include <avr/iotn84a.h>
#elif defined (__AVR_ATtiny841__)
#  include <avr/iotn841.h>
#elif defined (__AVR_ATtiny261__)
#  include <avr/iotn261.h>
#elif defined (__AVR_ATtiny261A__)
#  include <avr/iotn261a.h>
#elif defined (__AVR_ATtiny461__)
#  include <avr/iotn461.h>
#elif defined (__AVR_ATtiny461A__)
#  include <avr/iotn461a.h>
#elif defined (__AVR_ATtiny861__)
#  include <avr/iotn861.h>
#elif defined (__AVR_ATtiny861A__)
#  include <avr/iotn861a.h>
#elif defined (__AVR_ATtiny43U__)
#  include <avr/iotn43u.h>
#elif defined (__AVR_ATtiny48__)
#  include <avr/iotn48.h>
#elif defined (__AVR_ATtiny88__)
#  include <avr/iotn88.h>
#elif defined (__AVR_ATtiny828__)
#  include <avr/iotn828.h>
#elif defined (__AVR_ATtiny87__)
#  include <avr/iotn87.h>
#elif defined (__AVR_ATtiny167__)
#  include <avr/iotn167.h>
#elif defined (__AVR_ATtiny1634__)
#  include <avr/iotn1634.h>
#elif defined (__AVR_ATtiny202__)
#  include <avr/iotn202.h>
#elif defined (__AVR_ATtiny204__)
#  include <avr/iotn204.h>
#elif defined (__AVR_ATtiny212__)
#  include <avr/iotn212.h>
#elif defined (__AVR_ATtiny214__)
#  include <avr/iotn214.h>
#elif defined (__AVR_ATtiny402__)
#  include <avr/iotn402.h>
#elif defined (__AVR_ATtiny404__)
#  include <avr/iotn404.h>
#elif defined (__AVR_ATtiny406__)
#  include <avr/iotn406.h>
#elif defined (__AVR_ATtiny412__)
#  include <avr/iotn412.h>
#elif defined (__AVR_ATtiny414__)
#  include <avr/iotn414.h>
#elif defined (__AVR_ATtiny416__)
#  include <avr/iotn416.h>
#elif defined (__AVR_ATtiny417__)
#  include <avr/iotn417.h>
#elif defined (__AVR_ATtiny424__)
#  include <avr/iotn424.h>
#elif defined (__AVR_ATtiny426__)
#  include <avr/iotn426.h>
#elif defined (__AVR_ATtiny427__)
#  include <avr/iotn427.h>
#elif defined (__AVR_ATtiny804__)
#  include <avr/iotn804.h>
#elif defined (__AVR_ATtiny806__)
#  include <avr/iotn806.h>
#elif defined (__AVR_ATtiny807__)
#  include <avr/iotn807.h>
#elif defined (__AVR_ATtiny814__)
#  include <avr/iotn814.h>
#elif defined (__AVR_ATtiny816__)
#  include <avr/iotn816.h>
#elif defined (__AVR_ATtiny817__)
#  include <avr/iotn817.h>
#elif defined (__AVR_ATtiny824__)
#  include <avr/iotn824.h>
#elif defined (__AVR_ATtiny826__)
#  include <avr/iotn826.h>
#elif defined (__AVR_ATtiny827__)
#  include <avr/iotn827.h>
#elif defined (__AVR_ATtiny1604__)
#  include <avr/iotn1604.h>
```

```
#elif defined (__AVR_ATtiny1606__)
#  include <avr/iotn1606.h>
#elif defined (__AVR_ATtiny1607__)
#  include <avr/iotn1607.h>
#elif defined (__AVR_ATtiny1614__)
#  include <avr/iotn1614.h>
#elif defined (__AVR_ATtiny1616__)
#  include <avr/iotn1616.h>
#elif defined (__AVR_ATtiny1617__)
#  include <avr/iotn1617.h>
#elif defined (__AVR_ATtiny1624__)
#  include <avr/iotn1624.h>
#elif defined (__AVR_ATtiny1626__)
#  include <avr/iotn1626.h>
#elif defined (__AVR_ATtiny1627__)
#  include <avr/iotn1627.h>
#elif defined (__AVR_ATtiny3214__)
#  include <avr/iotn3214.h>
#elif defined (__AVR_ATtiny3216__)
#  include <avr/iotn3216.h>
#elif defined (__AVR_ATtiny3217__)
#  include <avr/iotn3217.h>
#elif defined (__AVR_ATtiny3224__)
#  include <avr/iotn3224.h>
#elif defined (__AVR_ATtiny3226__)
#  include <avr/iotn3226.h>
#elif defined (__AVR_ATtiny3227__)
#  include <avr/iotn3227.h>
#elif defined (__AVR_ATmega808__)
#  include <avr/iom808.h>
#elif defined (__AVR_ATmega809__)
#  include <avr/iom809.h>
#elif defined (__AVR_ATmega1608__)
#  include <avr/iom1608.h>
#elif defined (__AVR_ATmega1609__)
#  include <avr/iom1609.h>
#elif defined (__AVR_ATmega3208__)
#  include <avr/iom3208.h>
#elif defined (__AVR_ATmega3209__)
#  include <avr/iom3209.h>
#elif defined (__AVR_ATmega4808__)
#  include <avr/iom4808.h>
#elif defined (__AVR_ATmega4809__)
#  include <avr/iom4809.h>
#elif defined (__AVR_AT90SCR100__)
#  include <avr/io90scr100.h>
#elif defined (__AVR_ATxmega8E5__)
#  include <avr/iox8e5.h>
#elif defined (__AVR_ATxmega16A4__)
#  include <avr/iox16a4.h>
#elif defined (__AVR_ATxmega16A4U__)
#  include <avr/iox16a4u.h>
#elif defined (__AVR_ATxmega16C4__)
#  include <avr/iox16c4.h>
#elif defined (__AVR_ATxmega16D4__)
#  include <avr/iox16d4.h>
#elif defined (__AVR_ATxmega32A4__)
#  include <avr/iox32a4.h>
#elif defined (__AVR_ATxmega32A4U__)
#  include <avr/iox32a4u.h>
#elif defined (__AVR_ATxmega32C3__)
#  include <avr/iox32c3.h>
#elif defined (__AVR_ATxmega32C4__)
#  include <avr/iox32c4.h>
#elif defined (__AVR_ATxmega32D3__)
#  include <avr/iox32d3.h>
#elif defined (__AVR_ATxmega32D4__)
#  include <avr/iox32d4.h>
#elif defined (__AVR_ATxmega32E5__)
#  include <avr/iox32e5.h>
#elif defined (__AVR_ATxmega64A1__)
#  include <avr/iox64a1.h>
#elif defined (__AVR_ATxmega64A1U__)
#  include <avr/iox64a1u.h>
#elif defined (__AVR_ATxmega64A3__)
#  include <avr/iox64a3.h>
#elif defined (__AVR_ATxmega64A3U__)
#  include <avr/iox64a3u.h>
#elif defined (__AVR_ATxmega64A4U__)
#  include <avr/iox64a4u.h>
#elif defined (__AVR_ATxmega64B1__)
#  include <avr/iox64b1.h>
#elif defined (__AVR_ATxmega64B3__)
#  include <avr/iox64b3.h>
#elif defined (__AVR_ATxmega64C3__)
#  include <avr/iox64c3.h>
#elif defined (__AVR_ATxmega64D3__)
```

```
#  include <avr/iox64d3.h>
#elif defined (__AVR_ATxmega64D4__)
#  include <avr/iox64d4.h>
#elif defined (__AVR_ATxmega128A1__)
#  include <avr/iox128a1.h>
#elif defined (__AVR_ATxmega128A1U__)
#  include <avr/iox128a1u.h>
#elif defined (__AVR_ATxmega128A4U__)
#  include <avr/iox128a4u.h>
#elif defined (__AVR_ATxmega128A3__)
#  include <avr/iox128a3.h>
#elif defined (__AVR_ATxmega128A3U__)
#  include <avr/iox128a3u.h>
#elif defined (__AVR_ATxmega128B1__)
#  include <avr/iox128b1.h>
#elif defined (__AVR_ATxmega128B3__)
#  include <avr/iox128b3.h>
#elif defined (__AVR_ATxmega128C3__)
#  include <avr/iox128c3.h>
#elif defined (__AVR_ATxmega128D3__)
#  include <avr/iox128d3.h>
#elif defined (__AVR_ATxmega128D4__)
#  include <avr/iox128d4.h>
#elif defined (__AVR_ATxmega192A3__)
#  include <avr/iox192a3.h>
#elif defined (__AVR_ATxmega192A3U__)
#  include <avr/iox192a3u.h>
#elif defined (__AVR_ATxmega192C3__)
#  include <avr/iox192c3.h>
#elif defined (__AVR_ATxmega192D3__)
#  include <avr/iox192d3.h>
#elif defined (__AVR_ATxmega256A3__)
#  include <avr/iox256a3.h>
#elif defined (__AVR_ATxmega256A3U__)
#  include <avr/iox256a3u.h>
#elif defined (__AVR_ATxmega256A3B__)
#  include <avr/iox256a3b.h>
#elif defined (__AVR_ATxmega256A3BU__)
#  include <avr/iox256a3bu.h>
#elif defined (__AVR_ATxmega256C3__)
#  include <avr/iox256c3.h>
#elif defined (__AVR_ATxmega256D3__)
#  include <avr/iox256d3.h>
#elif defined (__AVR_ATxmega384C3__)
#  include <avr/iox384c3.h>
#elif defined (__AVR_ATxmega384D3__)
#  include <avr/iox384d3.h>
#elif defined (__AVR_ATA5702M322__)
#  include <avr/ioa5702m322.h>
#elif defined (__AVR_ATA5782__)
#  include <avr/ioa5782.h>
#elif defined (__AVR_ATA5790__)
#  include <avr/ioa5790.h>
#elif defined (__AVR_ATA5790N__)
#  include <avr/ioa5790n.h>
#elif defined (__AVR_ATA5831__)
#  include <avr/ioa5831.h>
#elif defined (__AVR_ATA5272__)
#  include <avr/ioa5272.h>
#elif defined (__AVR_ATA5505__)
#  include <avr/ioa5505.h>
#elif defined (__AVR_ATA5795__)
#  include <avr/ioa5795.h>
#elif defined (__AVR_ATA6285__)
#  include <avr/ioa6285.h>
#elif defined (__AVR_ATA6286__)
#  include <avr/ioa6286.h>
#elif defined (__AVR_ATA6289__)
#  include <avr/ioa6289.h>
#elif defined (__AVR_ATA6612C__)
#  include <avr/ioa6612c.h>
#elif defined (__AVR_ATA6613C__)
#  include <avr/ioa6613c.h>
#elif defined (__AVR_ATA6614Q__)
#  include <avr/ioa6614q.h>
#elif defined (__AVR_ATA6616C__)
#  include <avr/ioa6616c.h>
#elif defined (__AVR_ATA6617C__)
#  include <avr/ioa6617c.h>
#elif defined (__AVR_ATA664251__)
#  include <avr/ioa664251.h>
/* avr1: the following only supported for assembler programs */
#elif defined (__AVR_ATtiny28__)
#  include <avr/iotn28.h>
#elif defined (__AVR_AT90S1200__)
#  include <avr/io1200.h>
#elif defined (__AVR_ATtiny15__)
```

```
#  include <avr/iotn15.h>
#elif defined (__AVR_ATtiny12__)
#  include <avr/iotn12.h>
#elif defined (__AVR_ATtiny11__)
#  include <avr/iotn11.h>
#elif defined (__AVR_M3000__)
#  include <avr/iom3000.h>
#elif defined (__AVR_AVR32DA28__)
#  include <avr/ioavr32da28.h>
#elif defined (__AVR_AVR32DA32__)
#  include <avr/ioavr32da32.h>
#elif defined (__AVR_AVR32DA48__)
#  include <avr/ioavr32da48.h>
#elif defined (__AVR_AVR64DA28__)
#  include <avr/ioavr64da28.h>
#elif defined (__AVR_AVR64DA32__)
#  include <avr/ioavr64da32.h>
#elif defined (__AVR_AVR64DA48__)
#  include <avr/ioavr64da48.h>
#elif defined (__AVR_AVR64DA64__)
#  include <avr/ioavr64da64.h>
#elif defined (__AVR_AVR128DA28__)
#  include <avr/ioavr128da28.h>
#elif defined (__AVR_AVR128DA32__)
#  include <avr/ioavr128da32.h>
#elif defined (__AVR_AVR128DA48__)
#  include <avr/ioavr128da48.h>
#elif defined (__AVR_AVR128DA64__)
#  include <avr/ioavr128da64.h>
#elif defined (__AVR_AVR32DB28__)
#  include <avr/ioavr32db28.h>
#elif defined (__AVR_AVR32DB32__)
#  include <avr/ioavr32db32.h>
#elif defined (__AVR_AVR32DB48__)
#  include <avr/ioavr32db48.h>
#elif defined (__AVR_AVR64DB28__)
#  include <avr/ioavr64db28.h>
#elif defined (__AVR_AVR64DB32__)
#  include <avr/ioavr64db32.h>
#elif defined (__AVR_AVR64DB48__)
#  include <avr/ioavr64db48.h>
#elif defined (__AVR_AVR64DB64__)
#  include <avr/ioavr64db64.h>
#elif defined (__AVR_AVR128DB28__)
#  include <avr/ioavr128db28.h>
#elif defined (__AVR_AVR128DB32__)
#  include <avr/ioavr128db32.h>
#elif defined (__AVR_AVR128DB48__)
#  include <avr/ioavr128db48.h>
#elif defined (__AVR_AVR128DB64__)
#  include <avr/ioavr128db64.h>
#elif defined (__AVR_AVR16DD14__)
#  include <avr/ioavr16dd14.h>
#elif defined (__AVR_AVR16DD20__)
#  include <avr/ioavr16dd20.h>
#elif defined (__AVR_AVR16DD28__)
#  include <avr/ioavr16dd28.h>
#elif defined (__AVR_AVR16DD32__)
#  include <avr/ioavr16dd32.h>
#elif defined (__AVR_AVR32DD14__)
#  include <avr/ioavr32dd14.h>
#elif defined (__AVR_AVR32DD20__)
#  include <avr/ioavr32dd20.h>
#elif defined (__AVR_AVR32DD28__)
#  include <avr/ioavr32dd28.h>
#elif defined (__AVR_AVR32DD32__)
#  include <avr/ioavr32dd32.h>
#elif defined (__AVR_AVR64DD14__)
#  include <avr/ioavr64dd14.h>
#elif defined (__AVR_AVR64DD20__)
#  include <avr/ioavr64dd20.h>
#elif defined (__AVR_AVR64DD28__)
#  include <avr/ioavr64dd28.h>
#elif defined (__AVR_AVR64DD32__)
#  include <avr/ioavr64dd32.h>
#elif defined (__AVR_AVR64DU28__)
#  include <avr/ioavr64du28.h>
#elif defined (__AVR_AVR64DU32__)
#  include <avr/ioavr64du32.h>
#elif defined (__AVR_AVR16EA28__)
#  include <avr/ioavr16ea28.h>
#elif defined (__AVR_AVR16EA32__)
#  include <avr/ioavr16ea32.h>
#elif defined (__AVR_AVR16EA48__)
#  include <avr/ioavr16ea48.h>
#elif defined (__AVR_AVR16EB14__)
#  include <avr/ioavr16eb14.h>
```

```
#elif defined (__AVR_AVR16EB20__)
#  include <avr/ioavr16eb20.h>
#elif defined (__AVR_AVR16EB28__)
#  include <avr/ioavr16eb28.h>
#elif defined (__AVR_AVR16EB32__)
#  include <avr/ioavr16eb32.h>
#elif defined (__AVR_AVR32EA28__)
#  include <avr/ioavr32ea28.h>
#elif defined (__AVR_AVR32EA32__)
#  include <avr/ioavr32ea32.h>
#elif defined (__AVR_AVR32EA48__)
#  include <avr/ioavr32ea48.h>
#elif defined (__AVR_AVR64EA28__)
#  include <avr/ioavr64ea28.h>
#elif defined (__AVR_AVR64EA32__)
#  include <avr/ioavr64ea32.h>
#elif defined (__AVR_AVR64EA48__)
#  include <avr/ioavr64ea48.h>
#elif defined (__AVR_DEV_LIB_NAME__)
#  define __concat__(a,b) a##b
#  define __header1__(a,b) __concat__(a,b)
#  define __AVR_DEVICE_HEADER__ <avr/__header1__(io,__AVR_DEV_LIB_NAME__).h>
#  include __AVR_DEVICE_HEADER__
#else
#  if !defined(__COMPILING_AVR_LIBC__)
#    warning "device type not defined"
#  endif
#endif

#include <avr/portpins.h>

#include <avr/common.h>

#include <avr/version.h>

#if __AVR_ARCH__ >= 100
#  include <avr/xmega.h>
#endif

/* Include fuse.h after individual IO header files. */
#include <avr/fuse.h>

/* Include lock.h after individual IO header files. */
#include <avr/lock.h>

#endif /* _AVR_IO_H_ */
```

23.23 lock.h File Reference

23.24 lock.h

Go to the documentation of this file.

```
/* Copyright (c) 2007, Atmel Corporation
   All rights reserved.

   Redistribution and use in source and binary forms, with or without
   modification, are permitted provided that the following conditions are met:

   * Redistributions of source code must retain the above copyright
     notice, this list of conditions and the following disclaimer.

   * Redistributions in binary form must reproduce the above copyright
     notice, this list of conditions and the following disclaimer in
     the documentation and/or other materials provided with the
     distribution.

   * Neither the name of the copyright holders nor the names of
     contributors may be used to endorse or promote products derived
     from this software without specific prior written permission.

  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
  AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
  IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
  ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
  LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
  CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
  SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
  INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
  CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
  ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
```

```
  POSSIBILITY OF SUCH DAMAGE. */

/* $Id$ */

/* avr/lock.h - Lock Bits API */

#ifndef _AVR_LOCK_H_
#define _AVR_LOCK_H_  1


/** \file */
/** \defgroup avr_lock <avr/lock.h>: Lockbit Support

    \par Introduction

    The Lockbit API allows a user to specify the lockbit settings for the
    specific AVR device they are compiling for. These lockbit settings will be
    placed in a special section in the ELF output file, after linking.

    Programming tools can take advantage of the lockbit information embedded in
    the ELF file, by extracting this information and determining if the lockbits
    need to be programmed after programming the Flash and EEPROM memories.
    This also allows a single ELF file to contain all the
    information needed to program an AVR.

    To use the Lockbit API, include the <avr/io.h> header file, which in turn
    automatically includes the individual I/O header file and the <avr/lock.h>
    file. These other two files provides everything necessary to set the AVR
    lockbits.

    \par Lockbit API

    Each I/O header file may define up to 3 macros that controls what kinds
    of lockbits are available to the user.

    If __LOCK_BITS_EXIST is defined, then two lock bits are available to the
    user and 3 mode settings are defined for these two bits.

    If __BOOT_LOCK_BITS_0_EXIST is defined, then the two BLB0 lock bits are
    available to the user and 4 mode settings are defined for these two bits.

    If __BOOT_LOCK_BITS_1_EXIST is defined, then the two BLB1 lock bits are
    available to the user and 4 mode settings are defined for these two bits.

    If __BOOT_LOCK_APPLICATION_TABLE_BITS_EXIST is defined then two lock bits
    are available to set the locking mode for the Application Table Section
    (which is used in the XMEGA family).

    If __BOOT_LOCK_APPLICATION_BITS_EXIST is defined then two lock bits are
    available to set the locking mode for the Application Section (which is used
    in the XMEGA family).

    If __BOOT_LOCK_BOOT_BITS_EXIST is defined then two lock bits are available
    to set the locking mode for the Boot Loader Section (which is used in the
    XMEGA family).

    The AVR lockbit modes have inverted values, logical 1 for an unprogrammed
    (disabled) bit and logical 0 for a programmed (enabled) bit. The defined
    macros for each individual lock bit represent this in their definition by a
    bit-wise inversion of a mask. For example, the LB_MODE_3 macro is defined
    as:
    \code
    #define LB_MODE_3  (0xFC)
`   \endcode

    To combine the lockbit mode macros together to represent a whole byte,
    use the bitwise AND operator, like so:
    \code
    (LB_MODE_3 & BLB0_MODE_2)
    \endcode

    <avr/lock.h> also defines a macro that provides a default lockbit value:
    LOCKBITS_DEFAULT which is defined to be 0xFF.

    See the AVR device specific datasheet for more details about these
    lock bits and the available mode settings.

    A convenience macro, LOCKMEM, is defined as a GCC attribute for a
    custom-named section of ".lock".

    A convenience macro, LOCKBITS, is defined that declares a variable, __lock,
    of type unsigned char with the attribute defined by LOCKMEM. This variable
    allows the end user to easily set the lockbit data.

    \note If a device-specific I/O header file has previously defined LOCKMEM,
    then LOCKMEM is not redefined. If a device-specific I/O header file has
    previously defined LOCKBITS, then LOCKBITS is not redefined. LOCKBITS is
```

```
    currently known to be defined in the I/O header files for the XMEGA devices.

    \par API Usage Example

    Putting all of this together is easy:

    \code
    #include <avr/io.h>

    LOCKBITS = (LB_MODE_1 & BLB0_MODE_3 & BLB1_MODE_4);

    int main(void)
    {
        return 0;
    }
    \endcode

    Or:

    \code
    #include <avr/io.h>

    unsigned char __lock __attribute__((section (".lock"))) =
        (LB_MODE_1 & BLB0_MODE_3 & BLB1_MODE_4);

    int main(void)
    {
        return 0;
    }
    \endcode



    However there are a number of caveats that you need to be aware of to
    use this API properly.

    Be sure to include <avr/io.h> to get all of the definitions for the API.
    The LOCKBITS macro defines a global variable to store the lockbit data. This
    variable is assigned to its own linker section. Assign the desired lockbit
    values immediately in the variable initialization.

    The .lock section in the ELF file will get its values from the initial
    variable assignment ONLY. This means that you can NOT assign values to
    this variable in functions and the new values will not be put into the
    ELF .lock section.

    The global variable is declared in the LOCKBITS macro has two leading
    underscores, which means that it is reserved for the "implementation",
    meaning the library, so it will not conflict with a user-named variable.

    You must initialize the lockbit variable to some meaningful value, even
    if it is the default value. This is because the lockbits default to a
    logical 1, meaning unprogrammed. Normal uninitialized data defaults to all
    locgial zeros. So it is vital that all lockbits are initialized, even with
    default data. If they are not, then the lockbits may not programmed to the
    desired settings and can possibly put your device into an unrecoverable
    state.

    Be sure to have the -mmcu=<em>device</em> flag in your compile command line and
    your linker command line to have the correct device selected and to have
    the correct I/O header file included when you include <avr/io.h>.

    You can print out the contents of the .lock section in the ELF file by
    using this command line:
    \code
    avr-objdump -s -j .lock <ELF file>
    \endcode

*/


#if !(defined(__ASSEMBLER__) || defined(__DOXYGEN__))

#ifndef LOCKMEM
#define LOCKMEM  __attribute__((__used__, __section__ (".lock")))
#endif

#ifndef LOCKBITS
#define LOCKBITS unsigned char __lock LOCKMEM
#endif

/* Lock Bit Modes */
#if defined(__LOCK_BITS_EXIST)
#define LB_MODE_1  (0xFF)
#define LB_MODE_2  (0xFE)
#define LB_MODE_3  (0xFC)
#endif
```

```
#if defined(__BOOT_LOCK_BITS_0_EXIST)
#define BLB0_MODE_1  (0xFF)
#define BLB0_MODE_2  (0xFB)
#define BLB0_MODE_3  (0xF3)
#define BLB0_MODE_4  (0xF7)
#endif

#if defined(__BOOT_LOCK_BITS_1_EXIST)
#define BLB1_MODE_1  (0xFF)
#define BLB1_MODE_2  (0xEF)
#define BLB1_MODE_3  (0xCF)
#define BLB1_MODE_4  (0xDF)
#endif

#if defined(__BOOT_LOCK_APPLICATION_TABLE_BITS_EXIST)
#define BLBAT0 ~_BV(2)
#define BLBAT1 ~_BV(3)
#endif

#if defined(__BOOT_LOCK_APPLICATION_BITS_EXIST)
#define BLBA0 ~_BV(4)
#define BLBA1 ~_BV(5)
#endif

#if defined(__BOOT_LOCK_BOOT_BITS_EXIST)
#define BLBB0 ~_BV(6)
#define BLBB1 ~_BV(7)
#endif

#ifndef LOCKBITS_DEFAULT
#define LOCKBITS_DEFAULT (0xFF)
#endif

#endif  /* !(__ASSEMBLER || __DOXYGEN__) */


#endif /* _AVR_LOCK_H_ */
```

23.25 pgmspace.h File Reference

Macros

- #define PROGMEM_FAR __attribute__((__section__(".progmemx.data")))
- #define PROGMEM __attribute__((__progmem__))
- #define PSTR(str) ({ static const PROGMEM char c[ ] = (str); &c[0]; })
- #define PSTR_FAR(str) ({ static const PROGMEM_FAR char c[ ] = (str); pgm_get_far_address(c[0]); })
- #define pgm_read_byte_near(__addr) __LPM ((uint16_t)(__addr))
- #define pgm_read_word_near(__addr) __LPM_word ((uint16_t)(__addr))
- #define pgm_read_dword_near(__addr) __LPM_dword ((uint16_t)(__addr))
- #define pgm_read_qword_near(__addr) __LPM_qword ((uint16_t)(__addr))
- #define pgm_read_float_near(addr) pgm_read_float (addr)
- #define pgm_read_ptr_near(__addr) ((void∗) __LPM_word ((uint16_t)(__addr)))
- #define pgm_read_byte_far(__addr) __ELPM (__addr)
- #define pgm_read_word_far(__addr) __ELPM_word (__addr)
- #define pgm_read_dword_far(__addr) __ELPM_dword (__addr)
- #define pgm_read_qword_far(__addr) __ELPM_qword (__addr)
- #define pgm_read_ptr_far(__addr) ((void∗) __ELPM_word (__addr))
- #define pgm_read_byte(__addr) pgm_read_byte_near(__addr)
- #define pgm_read_word(__addr) pgm_read_word_near(__addr)
- #define pgm_read_dword(__addr) pgm_read_dword_near(__addr)
- #define pgm_read_qword(__addr) pgm_read_qword_near(__addr)
- #define pgm_read_ptr(__addr) pgm_read_ptr_near(__addr)
- #define pgm_get_far_address(var)

Functions

- static char pgm_read_char (const char ∗__addr)
- static unsigned char pgm_read_unsigned_char (const unsigned char ∗__addr)
- static signed char pgm_read_signed_char (const signed char ∗__addr)
- static uint8_t pgm_read_u8 (const uint8_t ∗__addr)
- static int8_t pgm_read_i8 (const int8_t ∗__addr)
- static short pgm_read_short (const short ∗__addr)
- static unsigned short pgm_read_unsigned_short (const unsigned short ∗__addr)
- static uint16_t pgm_read_u16 (const uint16_t ∗__addr)
- static int16_t pgm_read_i16 (const int16_t ∗__addr)
- static int pgm_read_int (const int ∗__addr)
- static signed pgm_read_signed (const signed ∗__addr)
- static unsigned pgm_read_unsigned (const unsigned ∗__addr)
- static signed int pgm_read_signed_int (const signed int ∗__addr)
- static unsigned int pgm_read_unsigned_int (const unsigned int ∗__addr)
- static __int24 pgm_read_i24 (const __int24 ∗__addr)
- static __uint24 pgm_read_u24 (const __uint24 ∗__addr)
- static uint32_t pgm_read_u32 (const uint32_t ∗__addr)
- static int32_t pgm_read_i32 (const int32_t ∗__addr)
- static long pgm_read_long (const long ∗__addr)
- static unsigned long pgm_read_unsigned_long (const unsigned long ∗__addr)
- static long long pgm_read_long_long (const long long ∗__addr)
- static unsigned long long pgm_read_unsigned_long_long (const unsigned long long ∗__addr)
- static uint64_t pgm_read_u64 (const uint64_t ∗__addr)
- static int64_t pgm_read_i64 (const int64_t ∗__addr)
- static float pgm_read_float (const float ∗__addr)
- static double pgm_read_double (const double ∗__addr)
- static long double pgm_read_long_double (const long double ∗__addr)
- static char pgm_read_char_far (uint_farptr_t __addr)
- static unsigned char pgm_read_unsigned_char_far (uint_farptr_t __addr)
- static signed char pgm_read_signed_char_far (uint_farptr_t __addr)
- static uint8_t pgm_read_u8_far (uint_farptr_t __addr)
- static int8_t pgm_read_i8_far (uint_farptr_t __addr)
- static short pgm_read_short_far (uint_farptr_t __addr)
- static unsigned short pgm_read_unsigned_short_far (uint_farptr_t __addr)
- static uint16_t pgm_read_u16_far (uint_farptr_t __addr)
- static int16_t pgm_read_i16_far (uint_farptr_t __addr)
- static int pgm_read_int_far (uint_farptr_t __addr)
- static unsigned pgm_read_unsigned_far (uint_farptr_t __addr)
- static unsigned int pgm_read_unsigned_int_far (uint_farptr_t __addr)
- static signed pgm_read_signed_far (uint_farptr_t __addr)
- static signed int pgm_read_signed_int_far (uint_farptr_t __addr)
- static long pgm_read_long_far (uint_farptr_t __addr)
- static unsigned long pgm_read_unsigned_long_far (uint_farptr_t __addr)
- static __int24 pgm_read_i24_far (uint_farptr_t __addr)
- static __uint24 pgm_read_u24_far (uint_farptr_t __addr)
- static uint32_t pgm_read_u32_far (uint_farptr_t __addr)
- static int32_t pgm_read_i32_far (uint_farptr_t __addr)
- static long long pgm_read_long_long_far (uint_farptr_t __addr)
- static unsigned long long pgm_read_unsigned_long_long_far (uint_farptr_t __addr)
- static uint64_t pgm_read_u64_far (uint_farptr_t __addr)
- static int64_t pgm_read_i64_far (uint_farptr_t __addr)
- static float pgm_read_float_far (uint_farptr_t __addr)
- static double pgm_read_double_far (uint_farptr_t __addr)

- static long double pgm_read_long_double_far (uint_farptr_t __addr)
- const void * memchr_P (const void *, int __val, size_t __len)
- int memcmp_P (const void *, const void *, size_t)
- void * memccpy_P (void *, const void *, int __val, size_t)
- void * memcpy_P (void *, const void *, size_t)
- void * memmem_P (const void *, size_t, const void *, size_t)
- const void * memrchr_P (const void *, int __val, size_t __len)
- char * strcat_P (char *, const char *)
- const char * strchr_P (const char *, int __val)
- const char * strchrnul_P (const char *, int __val)
- int strcmp_P (const char *, const char *)
- char * strcpy_P (char *, const char *)
- int strcasecmp_P (const char *, const char *)
- char * strcasestr_P (const char *, const char *)
- size_t strcspn_P (const char *__s, const char *__reject)
- size_t strlcat_P (char *, const char *, size_t)
- size_t strlcpy_P (char *, const char *, size_t)
- size_t strnlen_P (const char *, size_t)
- int strncmp_P (const char *, const char *, size_t)
- int strncasecmp_P (const char *, const char *, size_t)
- char * strncat_P (char *, const char *, size_t)
- char * strncpy_P (char *, const char *, size_t)
- char * strpbrk_P (const char *__s, const char *__accept)
- const char * strrchr_P (const char *, int __val)
- char * strsep_P (char **__sp, const char *__delim)
- size_t strspn_P (const char *__s, const char *__accept)
- char * strstr_P (const char *, const char *)
- char * strtok_P (char *__s, const char *__delim)
- char * strtok_rP (char *__s, const char *__delim, char **__last)
- size_t strlen_PF (uint_farptr_t src)
- size_t strnlen_PF (uint_farptr_t src, size_t len)
- void * memcpy_PF (void *dest, uint_farptr_t src, size_t len)
- char * strcpy_PF (char *dest, uint_farptr_t src)
- char * strncpy_PF (char *dest, uint_farptr_t src, size_t len)
- char * strcat_PF (char *dest, uint_farptr_t src)
- size_t strlcat_PF (char *dst, uint_farptr_t src, size_t siz)
- char * strncat_PF (char *dest, uint_farptr_t src, size_t len)
- int strcmp_PF (const char *s1, uint_farptr_t s2)
- int strncmp_PF (const char *s1, uint_farptr_t s2, size_t n)
- int strcasecmp_PF (const char *s1, uint_farptr_t s2)
- int strncasecmp_PF (const char *s1, uint_farptr_t s2, size_t n)
- uint_farptr_t strchr_PF (uint_farptr_t, int __val)
- char * strstr_PF (const char *s1, uint_farptr_t s2)
- size_t strlcpy_PF (char *dst, uint_farptr_t src, size_t siz)
- int memcmp_PF (const void *, uint_farptr_t, size_t)
- static size_t strlen_P (const char *s)
- template<typename T >
  T pgm_read< T > (const T *addr)
- template<typename T >
  T pgm_read_far< T > (uint_farptr_t addr)

23.26 pgmspace.h

Go to the documentation of this file.

```
/* Copyright (c) 2002-2007  Marek Michalkiewicz
   Copyright (c) 2006, Carlos Lamas
   Copyright (c) 2009-2010, Jan Waclawek
   All rights reserved.

   Redistribution and use in source and binary forms, with or without
   modification, are permitted provided that the following conditions are met:

   * Redistributions of source code must retain the above copyright
     notice, this list of conditions and the following disclaimer.
   * Redistributions in binary form must reproduce the above copyright
     notice, this list of conditions and the following disclaimer in
     the documentation and/or other materials provided with the
     distribution.
   * Neither the name of the copyright holders nor the names of
     contributors may be used to endorse or promote products derived
     from this software without specific prior written permission.

  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
  AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
  IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
  ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
  LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
  CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
  SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
  INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
  CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
  ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
  POSSIBILITY OF SUCH DAMAGE. */

/* $Id$ */

/*
   pgmspace.h

   Contributors:
     Created by Marek Michalkiewicz <marekm@linux.org.pl>
     Eric B. Weddington <eric@ecentral.com>
     Wolfgang Haidinger <wh@vmars.tuwien.ac.at> (pgm_read_dword())
     Ivanov Anton <anton@arc.com.ru> (pgm_read_float())
 */

/** \file */
/** \defgroup avr_pgmspace <avr/pgmspace.h>: Program Space Utilities
    \code
    #include <avr/io.h>
    #include <avr/pgmspace.h>
    \endcode

    The functions in this module provide interfaces for a program to access
    data stored in program space (flash memory) of the device.

    \note These functions are an attempt to provide some compatibility with
    header files that come with IAR C, to make porting applications between
    different compilers easier.  This is not 100% compatibility though (GCC
    does not have full support for multiple address spaces yet).

    \note If you are working with strings which are completely based in RAM,
    use the standard string functions described in \ref avr_string.

    \note If possible, put your constant tables in the lower 64 KB and use
    pgm_read_byte_near() or pgm_read_word_near() instead of
    pgm_read_byte_far() or pgm_read_word_far() since it is more efficient that
    way, and you can still use the upper 64K for executable code.
    All functions that are suffixed with a \c _P \e require their
    arguments to be in the lower 64 KB of the flash ROM, as they do
    not use ELPM instructions.  This is normally not a big concern as
    the linker setup arranges any program space constants declared
    using the macros from this header file so they are placed right after
    the interrupt vectors, and in front of any executable code.  However,
    it can become a problem if there are too many of these constants, or
    for bootloaders on devices with more than 64 KB of ROM.
    <em>All these functions will not work in that situation.</em>

    \note For <b>Xmega</b> devices, make sure the NVM controller
    command register (\c NVM.CMD or \c NVM_CMD) is set to 0x00 (NOP)
    before using any of these functions.
*/

#ifndef __PGMSPACE_H_
#define __PGMSPACE_H_ 1

#ifndef __DOXYGEN__
```

```
#define __need_size_t
#endif
#include <inttypes.h>
#include <stddef.h>
#include <avr/io.h>

#ifndef __DOXYGEN__
#ifndef __ATTR_CONST__
#define __ATTR_CONST__ __attribute__((__const__))
#endif

#ifndef __ATTR_PROGMEM__
#define __ATTR_PROGMEM__ __attribute__((__progmem__))
#endif

#ifndef __ATTR_PURE__
#define __ATTR_PURE__ __attribute__((__pure__))
#endif

#ifndef __ATTR_ALWAYS_INLINE__
#define __ATTR_ALWAYS_INLINE__ __inline__ __attribute__((__always_inline__))
#endif

#define PROGMEM __ATTR_PROGMEM__

#endif  /* !__DOXYGEN__ */

/**
   \ingroup avr_pgmspace
   \def PROGMEM_FAR

   Attribute to use in order to declare an object being located in
   far flash ROM.  This is similar to #PROGMEM, except that it puts
   the static storage object in section
   <tt>\ref sec_dot_progmemx ".progmemx.data"</tt>.
   In order to access the object,
   the <tt>pgm_read_*_far</tt> and \c _PF functions declare in this header
   can be used.  In order to get its address, see pgm_get_far_address().

   It only makes sense to put read-only objects in this section,
   though the compiler does not diagnose when this is not the case.  */
#define PROGMEM_FAR __attribute__((__section__(".progmemx.data")))

#ifdef __DOXYGEN__

/**
   \ingroup avr_pgmspace
   \def PROGMEM

   Attribute to use in order to declare an object being located in
   flash ROM.  */
#define PROGMEM __attribute__((__progmem__))

/** \ingroup avr_pgmspace
    \fn char pgm_read_char (const char *__addr)
    Read a <tt>char</tt> from 16-bit (near) byte-address \p __addr.
    The address is in the lower 64 KiB of program memory. */
static inline char pgm_read_char (const char *__addr);

/** \ingroup avr_pgmspace
    \fn unsigned char pgm_read_unsigned_char (const unsigned char *__addr)
    Read an <tt>unsigned char</tt> from 16-bit (near) byte-address \p __addr.
    The address is in the lower 64 KiB of program memory. */
static inline unsigned char pgm_read_unsigned_char (const unsigned char *__addr);

/** \ingroup avr_pgmspace
    \fn signed char pgm_read_signed_char (const signed char *__addr)
    Read a <tt>signed char</tt> from 16-bit (near) byte-address \p __addr.
    The address is in the lower 64 KiB of program memory. */
static inline signed char pgm_read_signed_char (const signed char *__addr);

/** \ingroup avr_pgmspace
    \fn uint8_t pgm_read_u8 (const uint8_t *__addr)
    Read an <tt>uint8_t</tt> from 16-bit (near) byte-address \p __addr.
    The address is in the lower 64 KiB of program memory. */
static inline uint8_t pgm_read_u8 (const uint8_t *__addr);

/** \ingroup avr_pgmspace
    \fn int8_t pgm_read_i8 (const int8_t *__addr)
    Read an <tt>int8_t</tt> from 16-bit (near) byte-address \p __addr.
    The address is in the lower 64 KiB of program memory. */
static inline int8_t pgm_read_i8 (const int8_t *__addr);

/** \ingroup avr_pgmspace
    \fn short pgm_read_short (const short *__addr)
    Read a <tt>short</tt> from 16-bit (near) byte-address \p __addr.
    The address is in the lower 64 KiB of program memory. */
```

```
static inline short pgm_read_short (const short *__addr);

/** \ingroup avr_pgmspace
    \fn unsigned short pgm_read_unsigned_short (const unsigned short *__addr)
    Read an <tt>unsigned short</tt> from 16-bit (near) byte-address \p __addr.
    The address is in the lower 64 KiB of program memory. */
static inline unsigned short pgm_read_unsigned_short (const unsigned short *__addr);

/** \ingroup avr_pgmspace
    \fn uint16_t pgm_read_u16 (const uint16_t *__addr)
    Read an <tt>uint16_t</tt> from 16-bit (near) byte-address \p __addr.
    The address is in the lower 64 KiB of program memory. */
static inline uint16_t pgm_read_u16 (const uint16_t *__addr);

/** \ingroup avr_pgmspace
    \fn int16_t pgm_read_i16 (const int16_t *__addr)
    Read an <tt>int16_t</tt> from 16-bit (near) byte-address \p __addr.
    The address is in the lower 64 KiB of program memory. */
static inline int16_t pgm_read_i16 (const int16_t *__addr);

/** \ingroup avr_pgmspace
    \fn int pgm_read_int (const int *__addr)
    Read an <tt>int</tt> from 16-bit (near) byte-address \p __addr.
    The address is in the lower 64 KiB of program memory. */
static inline int pgm_read_int (const int *__addr);

/** \ingroup avr_pgmspace
    \fn signed pgm_read_signed (const signed *__addr)
    Read a <tt>signed</tt> from 16-bit (near) byte-address \p __addr.
    The address is in the lower 64 KiB of program memory. */
static inline signed pgm_read_signed (const signed *__addr);

/** \ingroup avr_pgmspace
    \fn unsigned pgm_read_unsigned (const unsigned *__addr)
    Read an <tt>unsigned</tt> from 16-bit (near) byte-address \p __addr.
    The address is in the lower 64 KiB of program memory. */
static inline unsigned pgm_read_unsigned (const unsigned *__addr);

/** \ingroup avr_pgmspace
    \fn signed int pgm_read_signed_int (const signed int *__addr)
    Read a <tt>signed int</tt> from 16-bit (near) byte-address \p __addr.
    The address is in the lower 64 KiB of program memory. */
static inline signed int pgm_read_signed_int (const signed int *__addr);

/** \ingroup avr_pgmspace
    \fn unsigned int pgm_read_unsigned_int (const unsigned int *__addr)
    Read an <tt>unsigned int</tt> from 16-bit (near) byte-address \p __addr.
    The address is in the lower 64 KiB of program memory. */
static inline unsigned int pgm_read_unsigned_int (const unsigned int *__addr);

/** \ingroup avr_pgmspace
    \fn __int24 pgm_read_i24 (const __int24 *__addr)
    Read an <tt>__int24</tt> from 16-bit (near) byte-address \p __addr.
    The address is in the lower 64 KiB of program memory. */
static inline __int24 pgm_read_i24 (const __int24 *__addr);

/** \ingroup avr_pgmspace
    \fn __uint24 pgm_read_u24 (const __uint24 *__addr)
    Read an <tt>__uint24</tt> from 16-bit (near) byte-address \p __addr.
    The address is in the lower 64 KiB of program memory. */
static inline __uint24 pgm_read_u24 (const __uint24 *__addr);

/** \ingroup avr_pgmspace
    \fn uint32_t pgm_read_u32 (const uint32_t *__addr)
    Read an <tt>uint32_t</tt> from 16-bit (near) byte-address \p __addr.
    The address is in the lower 64 KiB of program memory. */
static inline uint32_t pgm_read_u32 (const uint32_t *__addr);

/** \ingroup avr_pgmspace
    \fn int32_t pgm_read_i32 (const int32_t *__addr)
    Read an <tt>int32_t</tt> from 16-bit (near) byte-address \p __addr.
    The address is in the lower 64 KiB of program memory. */
static inline int32_t pgm_read_i32 (const int32_t *__addr);

/** \ingroup avr_pgmspace
    \fn long pgm_read_long (const long *__addr)
    Read a <tt>long</tt> from 16-bit (near) byte-address \p __addr.
    The address is in the lower 64 KiB of program memory. */
static inline long pgm_read_long (const long *__addr);

/** \ingroup avr_pgmspace
    \fn unsigned long pgm_read_unsigned_long (const unsigned long *__addr)
    Read an <tt>unsigned long</tt> from 16-bit (near) byte-address \p __addr.
    The address is in the lower 64 KiB of program memory. */
static inline unsigned long pgm_read_unsigned_long (const unsigned long *__addr);

/** \ingroup avr_pgmspace
```

```
    \fn long long pgm_read_long_long (const long long *__addr)
    Read a <tt>long long</tt> from 16-bit (near) byte-address \p __addr.
    The address is in the lower 64 KiB of program memory. */
static inline long long pgm_read_long_long (const long long *__addr);

/** \ingroup avr_pgmspace
    \fn unsigned long long pgm_read_unsigned_long_long (const unsigned long long *__addr)
    Read an <tt>unsigned long long</tt> from 16-bit (near) byte-address
    \p __addr.
    The address is in the lower 64 KiB of program memory. */
static inline unsigned long long pgm_read_unsigned_long_long (const unsigned long long *__addr);

/** \ingroup avr_pgmspace
    \fn uint64_t pgm_read_u64 (const uint64_t *__addr)
    Read an <tt>uint64_t</tt> from 16-bit (near) byte-address \p __addr.
    The address is in the lower 64 KiB of program memory. */
static inline uint64_t pgm_read_u64 (const uint64_t *__addr);

/** \ingroup avr_pgmspace
    \fn int64_t pgm_read_i64 (const int64_t *__addr)
    Read an <tt>int64_t</tt> from 16-bit (near) byte-address \p __addr.
    The address is in the lower 64 KiB of program memory. */
static inline int64_t pgm_read_i64 (const int64_t *__addr);

/** \ingroup avr_pgmspace
    \fn float pgm_read_float (const float *__addr)
    Read a <tt>float</tt> from 16-bit (near) byte-address \p __addr.
    The address is in the lower 64 KiB of program memory. */
static inline float pgm_read_float (const float *__addr);

/** \ingroup avr_pgmspace
    \fn double pgm_read_double (const double *__addr)
    Read a <tt>double</tt> from 16-bit (near) byte-address \p __addr.
    The address is in the lower 64 KiB of program memory. */
static inline double pgm_read_double (const double *__addr);

/** \ingroup avr_pgmspace
    \fn long double pgm_read_long_double (const long double *__addr)
    Read a <tt>long double</tt> from 16-bit (near) byte-address \p __addr.
    The address is in the lower 64 KiB of program memory. */
static inline long double pgm_read_long_double (const long double *__addr);

#else /* !DOXYGEN */
#if defined(__AVR_TINY__)
/* For Reduced Tiny devices, avr-gcc adds 0x4000 when it takes the address
   of a PROGMEM object.  This means we can use open coded C/C++ to read
   from progmem.  This assumes we have
   - GCC PR71948 - Make progmem work on Reduced Tiny (GCC v7 / 2016-08)  */
#define __LPM__1(res, addr)  res = *addr
#define __LPM__2(res, addr)  res = *addr
#define __LPM__3(res, addr)  res = *addr
#define __LPM__4(res, addr)  res = *addr
#define __LPM__8(res, addr)  res = *addr

#elif defined(__AVR_HAVE_LPMX__)
#define __LPM__1(res, addr)                                     \
  __asm __volatile__ ("lpm %0,%a1"                              \
                      : "=r" (res) : "z" (addr))

#define __LPM__2(res, addr)                                     \
  __asm __volatile__ ("lpm %A0,%a1+"             "\n\t"         \
                      "lpm %B0,%a1+"                            \
                      : "=r" (res), "+z" (addr))

#define __LPM__3(res, addr)                                     \
  __asm __volatile__ ("lpm %A0,%a1+"             "\n\t"         \
                      "lpm %B0,%a1+"             "\n\t"         \
                      "lpm %C0,%a1+"                            \
                      : "=r" (res), "+z" (addr))

#define __LPM__4(res, addr)                                     \
  __asm __volatile__ ("lpm %A0,%a1+"             "\n\t"         \
                      "lpm %B0,%a1+"             "\n\t"         \
                      "lpm %C0,%a1+"             "\n\t"         \
                      "lpm %D0,%a1+"                            \
                      : "=r" (res), "+z" (addr))

#define __LPM__8(res, addr)                                     \
  __asm __volatile__ ("lpm %r0+0,%a1+"           "\n\t"         \
                      "lpm %r0+1,%a1+"           "\n\t"         \
                      "lpm %r0+2,%a1+"           "\n\t"         \
                      "lpm %r0+3,%a1+"           "\n\t"         \
                      "lpm %r0+4,%a1+"           "\n\t"         \
                      "lpm %r0+5,%a1+"           "\n\t"         \
                      "lpm %r0+6,%a1+"           "\n\t"         \
                      "lpm %r0+7,%a1+"                          \
                      : "=r" (res), "+z" (addr))
```

```
#else /* Has no LPMx and no Reduced Tiny => Has LPM.  */
#define __LPM__1(res, addr)                            \
  __asm __volatile__ ("lpm $ mov %A0,r0"               \
                      : "=r" (res) : "z" (addr) : "r0")

#define __LPM__2(res, addr)                                    \
  __asm __volatile__ ("lpm $ mov %A0,r0 $ adiw %1,1"  "\n\t"    \
                      "lpm $ mov %B0,r0"                        \
                      : "=r" (res), "+z" (addr) :: "r0")

#define __LPM__3(res, addr)                                    \
  __asm __volatile__ ("lpm $ mov %A0,r0 $ adiw %1,1"  "\n\t"    \
                      "lpm $ mov %B0,r0 $ adiw %1,1"  "\n\t"    \
                      "lpm $ mov %C0,r0"                        \
                      : "=r" (res), "+z" (addr) :: "r0")

#define __LPM__4(res, addr)                                    \
  __asm __volatile__ ("lpm $ mov %A0,r0 $ adiw %1,1"  "\n\t"    \
                      "lpm $ mov %B0,r0 $ adiw %1,1"  "\n\t"    \
                      "lpm $ mov %C0,r0 $ adiw %1,1"  "\n\t"    \
                      "lpm $ mov %D0,r0"                        \
                      : "=r" (res), "+z" (addr) :: "r0")

#define __LPM__8(res, addr)                                    \
  __asm __volatile__ ("lpm $ mov %r0+0,r0 $ adiw %1,1"  "\n\t"  \
                      "lpm $ mov %r0+1,r0 $ adiw %1,1"  "\n\t"  \
                      "lpm $ mov %r0+2,r0 $ adiw %1,1"  "\n\t"  \
                      "lpm $ mov %r0+3,r0 $ adiw %1,1"  "\n\t"  \
                      "lpm $ mov %r0+4,r0 $ adiw %1,1"  "\n\t"  \
                      "lpm $ mov %r0+5,r0 $ adiw %1,1"  "\n\t"  \
                      "lpm $ mov %r0+6,r0 $ adiw %1,1"  "\n\t"  \
                      "lpm $ mov %r0+7,r0"                      \
                      : "=r" (res), "+z" (addr) :: "r0")
#endif /* LPM cases */

#define _Avrlibc_Def_Pgm_1(Name, Typ)           \
  static __ATTR_ALWAYS_INLINE__                 \
  Typ pgm_read_##Name (const Typ *__addr)       \
  {                                             \
    Typ __res;                                  \
    __LPM__1 (__res, __addr);                   \
    return __res;                               \
  }

#define _Avrlibc_Def_Pgm_2(Name, Typ)           \
  static __ATTR_ALWAYS_INLINE__                 \
  Typ pgm_read_##Name (const Typ *__addr)       \
  {                                             \
    Typ __res;                                  \
    __LPM__2 (__res, __addr);                   \
    return __res;                               \
  }

#define _Avrlibc_Def_Pgm_3(Name, Typ)           \
  static __ATTR_ALWAYS_INLINE__                 \
  Typ pgm_read_##Name (const Typ *__addr)       \
  {                                             \
    Typ __res;                                  \
    __LPM__3 (__res, __addr);                   \
    return __res;                               \
  }

#define _Avrlibc_Def_Pgm_4(Name, Typ)           \
  static __ATTR_ALWAYS_INLINE__                 \
  Typ pgm_read_##Name (const Typ *__addr)       \
  {                                             \
    Typ __res;                                  \
    __LPM__4 (__res, __addr);                   \
    return __res;                               \
  }

#define _Avrlibc_Def_Pgm_8(Name, Typ)           \
  static __ATTR_ALWAYS_INLINE__                 \
  Typ pgm_read_##Name (const Typ *__addr)       \
  {                                             \
    Typ __res;                                  \
    __LPM__8 (__res, __addr);                   \
    return __res;                               \
  }

_Avrlibc_Def_Pgm_1 (char, char)
_Avrlibc_Def_Pgm_1 (unsigned_char, unsigned char)
_Avrlibc_Def_Pgm_1 (signed_char, signed char)
_Avrlibc_Def_Pgm_1 (u8, uint8_t)
_Avrlibc_Def_Pgm_1 (i8, int8_t)
#if __SIZEOF_INT__ == 1
_Avrlibc_Def_Pgm_1 (int, int)
```

```
_Avrlibc_Def_Pgm_1 (signed, signed)
_Avrlibc_Def_Pgm_1 (unsigned, unsigned)
_Avrlibc_Def_Pgm_1 (signed_int, signed int)
_Avrlibc_Def_Pgm_1 (unsigned_int, unsigned int)
#endif
#if __SIZEOF_SHORT__ == 1
_Avrlibc_Def_Pgm_1 (short, short)
_Avrlibc_Def_Pgm_1 (unsigned_short, unsigned short)
#endif

_Avrlibc_Def_Pgm_2 (u16, uint16_t)
_Avrlibc_Def_Pgm_2 (i16, int16_t)
#if __SIZEOF_INT__ == 2
_Avrlibc_Def_Pgm_2 (int, int)
_Avrlibc_Def_Pgm_2 (signed, signed)
_Avrlibc_Def_Pgm_2 (unsigned, unsigned)
_Avrlibc_Def_Pgm_2 (signed_int, signed int)
_Avrlibc_Def_Pgm_2 (unsigned_int, unsigned int)
#endif
#if __SIZEOF_SHORT__ == 2
_Avrlibc_Def_Pgm_2 (short, short)
_Avrlibc_Def_Pgm_2 (unsigned_short, unsigned short)
#endif
#if __SIZEOF_LONG__ == 2
_Avrlibc_Def_Pgm_2 (long, long)
_Avrlibc_Def_Pgm_2 (unsigned_long, unsigned long)
#endif

#if defined(__INT24_MAX__)
_Avrlibc_Def_Pgm_3 (i24, __int24)
_Avrlibc_Def_Pgm_3 (u24, __uint24)
#endif /* Have __int24 */

_Avrlibc_Def_Pgm_4 (u32, uint32_t)
_Avrlibc_Def_Pgm_4 (i32, int32_t)
_Avrlibc_Def_Pgm_4 (float, float)
#if __SIZEOF_LONG__ == 4
_Avrlibc_Def_Pgm_4 (long, long)
_Avrlibc_Def_Pgm_4 (unsigned_long, unsigned long)
#endif
#if __SIZEOF_LONG_LONG__ == 4
_Avrlibc_Def_Pgm_4 (long_long, long long)
_Avrlibc_Def_Pgm_4 (unsigned_long_long, unsigned long long)
#endif
#if __SIZEOF_DOUBLE__ == 4
_Avrlibc_Def_Pgm_4 (double, double)
#endif
#if __SIZEOF_LONG_DOUBLE__ == 4
_Avrlibc_Def_Pgm_4 (long_double, long double)
#endif

#if __SIZEOF_LONG_LONG__ == 8
_Avrlibc_Def_Pgm_8 (u64, uint64_t)
_Avrlibc_Def_Pgm_8 (i64, int64_t)
_Avrlibc_Def_Pgm_8 (long_long, long long)
_Avrlibc_Def_Pgm_8 (unsigned_long_long, unsigned long long)
#endif
#if __SIZEOF_DOUBLE__ == 8
_Avrlibc_Def_Pgm_8 (double, double)
#endif
#if __SIZEOF_LONG_DOUBLE__ == 8
_Avrlibc_Def_Pgm_8 (long_double, long double)
#endif

#endif /* DOXYGEN */

#ifdef __DOXYGEN__

/** \ingroup avr_pgmspace
    \fn char pgm_read_char_far (uint_farptr_t __addr)
    Read a <tt>char</tt> from far byte-address \p __addr.
    The address is in the program memory. */
static inline char pgm_read_char_far (uint_farptr_t __addr);

/** \ingroup avr_pgmspace
    \fn unsigned char pgm_read_unsigned_char_far (uint_farptr_t __addr)
    Read an <tt>unsigned char</tt> from far byte-address \p __addr.
    The address is in the program memory. */
static inline unsigned char pgm_read_unsigned_char_far (uint_farptr_t __addr);

/** \ingroup avr_pgmspace
    \fn signed char pgm_read_signed_char_far (uint_farptr_t __addr)
    Read a <tt>signed char</tt> from far byte-address \p __addr.
    The address is in the program memory. */
static inline signed char pgm_read_signed_char_far (uint_farptr_t __addr);

/** \ingroup avr_pgmspace
```

```
    \fn uint8_t pgm_read_u8_far (uint_farptr_t __addr)
    Read an <tt>uint8_t</tt> from far byte-address \p __addr.
    The address is in the program memory. */
static inline uint8_t pgm_read_u8_far (uint_farptr_t __addr);

/** \ingroup avr_pgmspace
    \fn int8_t pgm_read_i8_far (uint_farptr_t __addr)
    Read an <tt>int8_t</tt> from far byte-address \p __addr.
    The address is in the program memory. */
static inline int8_t pgm_read_i8_far (uint_farptr_t __addr);

/** \ingroup avr_pgmspace
    \fn short pgm_read_short_far (uint_farptr_t __addr)
    Read a <tt>short</tt> from far byte-address \p __addr.
    The address is in the program memory. */
static inline short pgm_read_short_far (uint_farptr_t __addr);

/** \ingroup avr_pgmspace
    \fn unsigned short pgm_read_unsigned_short_far (uint_farptr_t __addr)
    Read an <tt>unsigned short</tt> from far byte-address \p __addr.
    The address is in the program memory. */
static inline unsigned short pgm_read_unsigned_short_far (uint_farptr_t __addr);

/** \ingroup avr_pgmspace
    \fn uint16_t pgm_read_u16_far (uint_farptr_t __addr)
    Read an <tt>uint16_t</tt> from far byte-address \p __addr.
    The address is in the program memory. */
static inline uint16_t pgm_read_u16_far (uint_farptr_t __addr);

/** \ingroup avr_pgmspace
    \fn int16_t pgm_read_i16_far (uint_farptr_t __addr)
    Read an <tt>int16_t</tt> from far byte-address \p __addr.
    The address is in the program memory. */
static inline int16_t pgm_read_i16_far (uint_farptr_t __addr);

/** \ingroup avr_pgmspace
    \fn int pgm_read_int_far (uint_farptr_t __addr)
    Read an <tt>int</tt> from far byte-address \p __addr.
    The address is in the program memory. */
static inline int pgm_read_int_far (uint_farptr_t __addr);

/** \ingroup avr_pgmspace
    \fn unsigned pgm_read_unsigned_far (uint_farptr_t __addr)
    Read an <tt>unsigned</tt> from far byte-address \p __addr.
    The address is in the program memory. */
static inline unsigned pgm_read_unsigned_far (uint_farptr_t __addr);

/** \ingroup avr_pgmspace
    \fn unsigned int pgm_read_unsigned_int_far (uint_farptr_t __addr)
    Read an <tt>unsigned int</tt> from far byte-address \p __addr.
    The address is in the program memory. */
static inline unsigned int pgm_read_unsigned_int_far (uint_farptr_t __addr);

/** \ingroup avr_pgmspace
    \fn signed pgm_read_signed_far (uint_farptr_t __addr)
    Read a <tt>signed</tt> from far byte-address \p __addr.
    The address is in the program memory. */
static inline signed pgm_read_signed_far (uint_farptr_t __addr);

/** \ingroup avr_pgmspace
    \fn signed int pgm_read_signed_int_far (uint_farptr_t __addr)
    Read a <tt>signed int</tt> from far byte-address \p __addr.
    The address is in the program memory. */
static inline signed int pgm_read_signed_int_far (uint_farptr_t __addr);

/** \ingroup avr_pgmspace
    \fn long pgm_read_long_far (uint_farptr_t __addr)
    Read a <tt>long</tt> from far byte-address \p __addr.
    The address is in the program memory. */
static inline long pgm_read_long_far (uint_farptr_t __addr);

/** \ingroup avr_pgmspace
    \fn unsigned long pgm_read_unsigned_long_far (uint_farptr_t __addr)
    Read an <tt>unsigned long</tt> from far byte-address \p __addr.
    The address is in the program memory. */
static inline unsigned long pgm_read_unsigned_long_far (uint_farptr_t __addr);

/** \ingroup avr_pgmspace
    \fn __int24 pgm_read_i24_far (uint_farptr_t __addr)
    Read an <tt>__int24</tt> from far byte-address \p __addr.
    The address is in the program memory. */
static inline __int24 pgm_read_i24_far (uint_farptr_t __addr);

/** \ingroup avr_pgmspace
    \fn __uint24 pgm_read_u24_far (uint_farptr_t __addr)
    Read an <tt>__uint24</tt> from far byte-address \p __addr.
    The address is in the program memory. */
```

```
static inline __uint24 pgm_read_u24_far (uint_farptr_t __addr);

/** \ingroup avr_pgmspace
    \fn uint32_t pgm_read_u32_far (uint_farptr_t __addr)
    Read an <tt>uint32_t</tt> from far byte-address \p __addr.
    The address is in the program memory. */
static inline uint32_t pgm_read_u32_far (uint_farptr_t __addr);

/** \ingroup avr_pgmspace
    \fn int32_t pgm_read_i32_far (uint_farptr_t __addr)
    Read an <tt>int32_t</tt> from far byte-address \p __addr.
    The address is in the program memory. */
static inline int32_t pgm_read_i32_far (uint_farptr_t __addr);

/** \ingroup avr_pgmspace
    \fn long long pgm_read_long_long_far (uint_farptr_t __addr)
    Read a <tt>long long</tt> from far byte-address \p __addr.
    The address is in the program memory. */
static inline long long pgm_read_long_long_far (uint_farptr_t __addr);

/** \ingroup avr_pgmspace
    \fn unsigned long long pgm_read_unsigned_long_long_far (uint_farptr_t __addr)
    Read an <tt>unsigned long long</tt> from far byte-address \p __addr.
    The address is in the program memory. */
static inline unsigned long long pgm_read_unsigned_long_long_far (uint_farptr_t __addr);

/** \ingroup avr_pgmspace
    \fn uint64_t pgm_read_u64_far (uint_farptr_t __addr)
    Read an <tt>uint64_t</tt> from far byte-address \p __addr.
    The address is in the program memory. */
static inline uint64_t pgm_read_u64_far (uint_farptr_t __addr);

/** \ingroup avr_pgmspace
    \fn int64_t pgm_read_i64_far (uint_farptr_t __addr)
    Read an <tt>int64_t</tt> from far byte-address \p __addr.
    The address is in the program memory. */
static inline int64_t pgm_read_i64_far (uint_farptr_t __addr);

/** \ingroup avr_pgmspace
    \fn float pgm_read_float_far (uint_farptr_t __addr)
    Read a <tt>float</tt> from far byte-address \p __addr.
    The address is in the program memory. */
static inline float pgm_read_float_far (uint_farptr_t __addr);

/** \ingroup avr_pgmspace
    \fn double pgm_read_double_far (uint_farptr_t __addr)
    Read a <tt>double</tt> from far byte-address \p __addr.
    The address is in the program memory. */
static inline double pgm_read_double_far (uint_farptr_t __addr);

/** \ingroup avr_pgmspace
    \fn long double pgm_read_long_double_far (uint_farptr_t __addr)
    Read a <tt>long double</tt> from far byte-address \p __addr.
    The address is in the program memory. */
static inline long double pgm_read_long_double_far (uint_farptr_t __addr);

#else /* !DOXYGEN */

#if defined(__AVR_HAVE_ELPMX__)

#ifdef __AVR_HAVE_RAMPD__
/* For devices with EBI, reset RAMPZ to zero after.  */
#define __pgm_clr_RAMPZ_ "\n\t" "out  %i2,__zero_reg__"
#else
/* Devices without EBI: no need to reset RAMPZ.  */
#define __pgm_clr_RAMPZ_ /* empty */
#endif

#define __ELPM__1(res, addr, T)                              \
  __asm __volatile__ ("movw r30,%1"          "\n\t"      \
                      "out  %i2,%C1"         "\n\t"      \
                      "elpm %A0,Z"                       \
                      __pgm_clr_RAMPZ_                   \
                      : "=r" (res)                       \
                      : "r" (addr), "n" (& RAMPZ)        \
                      : "r30", "r31")

#define __ELPM__2(res, addr, T)                              \
  __asm __volatile__ ("movw r30,%1"          "\n\t"      \
                      "out  %i2,%C1"         "\n\t"      \
                      "elpm %A0,Z+"          "\n\t"      \
                      "elpm %B0,Z+"                      \
                      __pgm_clr_RAMPZ_                   \
                      : "=r" (res)                       \
                      : "r" (addr), "n" (& RAMPZ)        \
                      : "r30", "r31")

```

```
#define __ELPM__3(res, addr, T)                                 \
  __asm __volatile__ ("movw r30,%1"         "\n\t"      \
                      "out  %i2,%C1"        "\n\t"      \
                      "elpm %A0,Z+"         "\n\t"      \
                      "elpm %B0,Z+"         "\n\t"      \
                      "elpm %C0,Z+"                     \
                      __pgm_clr_RAMPZ_                  \
                      : "=r" (res)                      \
                      : "r" (addr), "n" (& RAMPZ)       \
                      : "r30", "r31")

#define __ELPM__4(res, addr, T)                                 \
  __asm __volatile__ ("movw r30,%1"         "\n\t"      \
                      "out  %i2,%C1"        "\n\t"      \
                      "elpm %A0,Z+"         "\n\t"      \
                      "elpm %B0,Z+"         "\n\t"      \
                      "elpm %C0,Z+"         "\n\t"      \
                      "elpm %D0,Z+"                     \
                      __pgm_clr_RAMPZ_                  \
                      : "=r" (res)                      \
                      : "r" (addr), "n" (& RAMPZ)       \
                      : "r30", "r31")

#define __ELPM__8(res, addr, T)                                 \
  __asm __volatile__ ("movw r30,%1"         "\n\t"      \
                      "out  %i2,%C1"        "\n\t"      \
                      "elpm %r0+0,Z+"       "\n\t"      \
                      "elpm %r0+1,Z+"       "\n\t"      \
                      "elpm %r0+2,Z+"       "\n\t"      \
                      "elpm %r0+3,Z+"       "\n\t"      \
                      "elpm %r0+4,Z+"       "\n\t"      \
                      "elpm %r0+5,Z+"       "\n\t"      \
                      "elpm %r0+6,Z+"       "\n\t"      \
                      "elpm %r0+7,Z+"                   \
                      __pgm_clr_RAMPZ_                  \
                      : "=r" (res)                      \
                      : "r" (addr), "n" (& RAMPZ)       \
                      : "r30", "r31")

/* FIXME: AT43USB320 does not have RAMPZ but supports (external) program
   memory of 64 KiW, at least that's what the comments in io43usb32x.h are
   indicating.  A solution would be to put the device in a different
   multilib-set (see GCC PR78275), as io.h has  "#define FLASHEND 0x0FFFF".
   For now, just exclude AT43USB320 from code that uses RAMPZ. Also note
   that the manual asserts that the entire program memory can be accessed
   by LPM, implying only 64 KiB of program memory.  */
#elif defined(__AVR_HAVE_ELPM__) \
      && !defined(__AVR_AT43USB320__)
/* The poor devices without ELPMx: Do 24-bit addresses by hand... */
#define __ELPM__1(res, addr, T)                                         \
  __asm __volatile__ ("mov r30,%A1"     "\n\t"                          \
                      "mov r31,%B1"     "\n\t"                          \
                      "out %i2,%C1 $ elpm $ mov %A0,r0"                 \
                      : "=r" (res)                                      \
                      : "r" (addr), "n" (& RAMPZ)                       \
                      : "r30", "r31", "r0")

#define __ELPM__2(res, addr, T)                                         \
  __asm __volatile__                                                    \
   ("mov r30,%A1"     "\n\t"                                            \
    "mov r31,%B1"     "\n\t"                                            \
    "mov %B0,%C1"     "\n\t"                                            \
    "out %i2,%B0 $ elpm $ mov %A0,r0 $ adiw r30,1 $ adc %B0,r1" "\n\t"  \
    "out %i2,%B0 $ elpm $ mov %B0,r0"                                   \
    : "=r" (res)                                                        \
    : "r" (addr), "n" (& RAMPZ)                                         \
    : "r30", "r31", "r0")

#define __ELPM__3(res, addr, T)                                         \
  __asm __volatile__                                                    \
   ("mov r30,%A1"     "\n\t"                                            \
    "mov r31,%B1"     "\n\t"                                            \
    "mov %C0,%C1"     "\n\t"                                            \
    "out %i2,%C0 $ elpm $ mov %A0,r0 $ adiw r30,1 $ adc %C0,r1" "\n\t"  \
    "out %i2,%C0 $ elpm $ mov %B0,r0 $ adiw r30,1 $ adc %C0,r1" "\n\t"  \
    "out %i2,%C0 $ elpm $ mov %C0,r0"                                   \
    : "=r" (res)                                                        \
    : "r" (addr), "n" (& RAMPZ)                                         \
    : "r30", "r31", "r0")

#define __ELPM__4(res, addr, T)                                         \
  __asm __volatile__                                                    \
   ("mov r30,%A1"     "\n\t"                                            \
    "mov r31,%B1"     "\n\t"                                            \
    "mov %D0,%C1"     "\n\t"                                            \
    "out %i2,%D0 $ elpm $ mov %A0,r0 $ adiw r30,1 $ adc %D0,r1" "\n\t"  \
    "out %i2,%D0 $ elpm $ mov %B0,r0 $ adiw r30,1 $ adc %D0,r1" "\n\t"  \
```

```
    "out %i2,%D0 $ elpm $ mov %C0,r0 $ adiw r30,1 $ adc %D0,r1" "\n\t"   \
    "out %i2,%D0 $ elpm $ mov %D0,r0"                                     \
    : "=r" (res)                                                          \
    : "r" (addr), "n" (& RAMPZ)                                           \
    : "r30", "r31", "r0")

#define __ELPM__8(res, addr, T)                                           \
  __asm __volatile__                                                      \
  ("mov r30,%A1"    "\n\t"                                                \
   "mov r31,%B1"    "\n\t"                                                \
   "mov %r0+7,%C1"  "\n\t"                                                \
   "out %i2,%r0+7 $ elpm $ mov %r0+0,r0 $ adiw r30,1 $ adc %r0+7,r1" "\n\t" \
   "out %i2,%r0+7 $ elpm $ mov %r0+1,r0 $ adiw r30,1 $ adc %r0+7,r1" "\n\t" \
   "out %i2,%r0+7 $ elpm $ mov %r0+2,r0 $ adiw r30,1 $ adc %r0+7,r1" "\n\t" \
   "out %i2,%r0+7 $ elpm $ mov %r0+3,r0 $ adiw r30,1 $ adc %r0+7,r1" "\n\t" \
   "out %i2,%r0+7 $ elpm $ mov %r0+4,r0 $ adiw r30,1 $ adc %r0+7,r1" "\n\t" \
   "out %i2,%r0+7 $ elpm $ mov %r0+5,r0 $ adiw r30,1 $ adc %r0+7,r1" "\n\t" \
   "out %i2,%r0+7 $ elpm $ mov %r0+6,r0 $ adiw r30,1 $ adc %r0+7,r1" "\n\t" \
   "out %i2,%r0+7 $ elpm $ mov %r0+7,r0"                                  \
   : "=r" (res)                                                           \
   : "r" (addr), "n" (& RAMPZ)                                            \
   : "r30", "r31", "r0")
#else
/* No ELPM: Fall back to __LPM__<N>.  */
#define __ELPM__1(r,a,T) const T *__a = (const T*)(uint16_t) a; __LPM__1(r,__a)
#define __ELPM__2(r,a,T) const T *__a = (const T*)(uint16_t) a; __LPM__2(r,__a)
#define __ELPM__3(r,a,T) const T *__a = (const T*)(uint16_t) a; __LPM__3(r,__a)
#define __ELPM__4(r,a,T) const T *__a = (const T*)(uint16_t) a; __LPM__4(r,__a)
#define __ELPM__8(r,a,T) const T *__a = (const T*)(uint16_t) a; __LPM__8(r,__a)
#endif /* ELPM cases */

#define _Avrlibc_Def_Pgm_Far_1(Name, Typ)           \
  static __ATTR_ALWAYS_INLINE__                     \
  Typ pgm_read_##Name##_far (uint_farptr_t __addr) \
  {                                                 \
    Typ __res;                                      \
    __ELPM__1 (__res, __addr, Typ);                 \
    return __res;                                   \
  }

#define _Avrlibc_Def_Pgm_Far_2(Name, Typ)           \
  static __ATTR_ALWAYS_INLINE__                     \
  Typ pgm_read_##Name##_far (uint_farptr_t __addr) \
  {                                                 \
    Typ __res;                                      \
    __ELPM__2 (__res, __addr, Typ);                 \
    return __res;                                   \
  }

#define _Avrlibc_Def_Pgm_Far_3(Name, Typ)           \
  static __ATTR_ALWAYS_INLINE__                     \
  Typ pgm_read_##Name##_far (uint_farptr_t __addr) \
  {                                                 \
    Typ __res;                                      \
    __ELPM__3 (__res, __addr, Typ);                 \
    return __res;                                   \
  }

#define _Avrlibc_Def_Pgm_Far_4(Name, Typ)           \
  static __ATTR_ALWAYS_INLINE__                     \
  Typ pgm_read_##Name##_far (uint_farptr_t __addr) \
  {                                                 \
    Typ __res;                                      \
    __ELPM__4 (__res, __addr, Typ);                 \
    return __res;                                   \
  }

#define _Avrlibc_Def_Pgm_Far_8(Name, Typ)           \
  static __ATTR_ALWAYS_INLINE__                     \
  Typ pgm_read_##Name##_far (uint_farptr_t __addr) \
  {                                                 \
    Typ __res;                                      \
    __ELPM__8 (__res, __addr, Typ);                 \
    return __res;                                   \
  }

_Avrlibc_Def_Pgm_Far_1 (char, char)
_Avrlibc_Def_Pgm_Far_1 (unsigned_char, unsigned char)
_Avrlibc_Def_Pgm_Far_1 (signed_char, signed char)
_Avrlibc_Def_Pgm_Far_1 (u8, uint8_t)
_Avrlibc_Def_Pgm_Far_1 (i8, int8_t)
#if __SIZEOF_INT__ == 1
_Avrlibc_Def_Pgm_Far_1 (int, int)
_Avrlibc_Def_Pgm_Far_1 (unsigned, unsigned)
_Avrlibc_Def_Pgm_Far_1 (unsigned_int, unsigned int)
_Avrlibc_Def_Pgm_Far_1 (signed, signed)
_Avrlibc_Def_Pgm_Far_1 (signed_int, signed int)
```

```
#endif
#if __SIZEOF_SHORT__ == 1
_Avrlibc_Def_Pgm_Far_1 (short, short)
_Avrlibc_Def_Pgm_Far_1 (unsigned_short, unsigned short)
#endif

_Avrlibc_Def_Pgm_Far_2 (u16, uint16_t)
_Avrlibc_Def_Pgm_Far_2 (i16, int16_t)
#if __SIZEOF_INT__ == 2
_Avrlibc_Def_Pgm_Far_2 (int, int)
_Avrlibc_Def_Pgm_Far_2 (unsigned, unsigned)
_Avrlibc_Def_Pgm_Far_2 (unsigned_int, unsigned int)
_Avrlibc_Def_Pgm_Far_2 (signed, signed)
_Avrlibc_Def_Pgm_Far_2 (signed_int, signed int)
#endif
#if __SIZEOF_SHORT__ == 2
_Avrlibc_Def_Pgm_Far_2 (short, short)
_Avrlibc_Def_Pgm_Far_2 (unsigned_short, unsigned short)
#endif
#if __SIZEOF_LONG__ == 2
_Avrlibc_Def_Pgm_Far_2 (long, long)
_Avrlibc_Def_Pgm_Far_2 (unsigned_long, unsigned long)
#endif

#if defined(__INT24_MAX__)
_Avrlibc_Def_Pgm_Far_3 (i24, __int24)
_Avrlibc_Def_Pgm_Far_3 (u24, __uint24)
#endif /* Have __int24 */

_Avrlibc_Def_Pgm_Far_4 (u32, uint32_t)
_Avrlibc_Def_Pgm_Far_4 (i32, int32_t)
_Avrlibc_Def_Pgm_Far_4 (float, float)
#if __SIZEOF_LONG__ == 4
_Avrlibc_Def_Pgm_Far_4 (long, long)
_Avrlibc_Def_Pgm_Far_4 (unsigned_long, unsigned long)
#endif
#if __SIZEOF_LONG_LONG__ == 4
_Avrlibc_Def_Pgm_Far_4 (long_long, long long)
_Avrlibc_Def_Pgm_Far_4 (unsigned_long_long, unsigned long long)
#endif
#if __SIZEOF_DOUBLE__ == 4
_Avrlibc_Def_Pgm_Far_4 (double, double)
#endif
#if __SIZEOF_LONG_DOUBLE__ == 4
_Avrlibc_Def_Pgm_Far_4 (long_double, long double)
#endif

#if __SIZEOF_LONG_LONG__ == 8
_Avrlibc_Def_Pgm_Far_8 (u64, uint64_t)
_Avrlibc_Def_Pgm_Far_8 (i64, int64_t)
_Avrlibc_Def_Pgm_Far_8 (long_long, long long)
_Avrlibc_Def_Pgm_Far_8 (unsigned_long_long, unsigned long long)
#endif
#if __SIZEOF_DOUBLE__ == 8
_Avrlibc_Def_Pgm_Far_8 (double, double)
#endif
#if __SIZEOF_LONG_DOUBLE__ == 8
_Avrlibc_Def_Pgm_Far_8 (long_double, long double)
#endif

#endif /* DOXYGEN */

#ifdef __cplusplus
extern "C" {
#endif

#if defined(__DOXYGEN__)
/* No documentation for the deprecated stuff.  */
#elif defined(__PROG_TYPES_COMPAT__)  /* !DOXYGEN */

typedef void prog_void __attribute__((__progmem__,__deprecated__("prog_void type is deprecated.")));
typedef char prog_char __attribute__((__progmem__,__deprecated__("prog_char type is deprecated.")));
typedef unsigned char prog_uchar __attribute__((__progmem__,__deprecated__("prog_uchar type is
      deprecated.")));
typedef int8_t    prog_int8_t   __attribute__((__progmem__,__deprecated__("prog_int8_t type is
      deprecated.")));
typedef uint8_t   prog_uint8_t  __attribute__((__progmem__,__deprecated__("prog_uint8_t type is
      deprecated.")));
typedef int16_t   prog_int16_t  __attribute__((__progmem__,__deprecated__("prog_int16_t type is
      deprecated.")));
typedef uint16_t  prog_uint16_t __attribute__((__progmem__,__deprecated__("prog_uint16_t type is
      deprecated.")));
typedef int32_t   prog_int32_t  __attribute__((__progmem__,__deprecated__("prog_int32_t type is
      deprecated.")));
typedef uint32_t  prog_uint32_t __attribute__((__progmem__,__deprecated__("prog_uint32_t type is
      deprecated.")));
#if !__USING_MINT8
```

```
typedef int64_t   prog_int64_t  __attribute__((__progmem__,__deprecated__("prog_int64_t type is deprecated.")));
typedef uint64_t  prog_uint64_t __attribute__((__progmem__,__deprecated__("prog_uint64_t type is deprecated.")));
#endif

#ifndef PGM_P
#define PGM_P const prog_char *
#endif

#ifndef PGM_VOID_P
#define PGM_VOID_P const prog_void *
#endif

#else /* !defined(__DOXYGEN__), !defined(__PROG_TYPES_COMPAT__) */

#ifndef PGM_P
#define PGM_P const char *
#endif

#ifndef PGM_VOID_P
#define PGM_VOID_P const void *
#endif
#endif /* defined(__DOXYGEN__), defined(__PROG_TYPES_COMPAT__) */

/* Although in C, we can get away with just using __c, it does not work in
   C++. We need to use &__c[0] to avoid the compiler puking. Dave Hylands
   explaned it thusly,

     Let's suppose that we use PSTR("Test"). In this case, the type returned
     by __c is a prog_char[5] and not a prog_char *. While these are
     compatible, they aren't the same thing (especially in C++). The type
     returned by &__c[0] is a prog_char *, which explains why it works
     fine. */

#if defined(__DOXYGEN__)
/*
 * The #define below is just a dummy that serves documentation
 * purposes only.
 */
/** \ingroup avr_pgmspace
    \def PSTR(str)

    Used to declare a static pointer to a string in program space. */
# define PSTR(str) ({ static const PROGMEM char c[] = (str); &c[0]; })
#else  /* !DOXYGEN */
/* The real thing. */
# define PSTR(s) (__extension__({static const char __c[] PROGMEM = (s); &__c[0];}))
#endif /* DOXYGEN */

#if defined(__DOXYGEN__)
/** \ingroup avr_pgmspace
    \def PSTR_FAR(str)

    Used to define a string literal in far program space, and to return its
    address of type #uint_farptr_t. */
# define PSTR_FAR(str) ({ static const PROGMEM_FAR char c[] = (str); pgm_get_far_address(c[0]); })
#else  /* !DOXYGEN */
/* The real thing. */
# define PSTR_FAR(s) (__extension__({static const char __c[] PROGMEM_FAR = (s); pgm_get_far_address(__c[0]);}))
#endif /* DOXYGEN */

#ifndef __DOXYGEN__

/* These are used down the line for pgm_read_byte[_near] etc. */

#if defined (__AVR_TINY__)
/* Attribute __progmem__ on Reduced Tiny works different than for
   all the other devices:  When taking the address of a symbol that's
   attributed as progmem, then the compiler adds an offset of 0x4000
   to the value of the symbol.  This means that accessing data in
   progmem can be performed by vanilla C/C++ code.  This requires
   - GCC PR71948 - Make progmem work on Reduced Tiny (GCC v7 / 2016-08)  */
#define __LPM(addr)         (* (const uint8_t*)(addr))
#define __LPM_word(addr)    (* (const uint16_t*)(addr))
#define __LPM_dword(addr)   (* (const uint32_t*)(addr))
#  if __SIZEOF_LONG_LONG__ == 8
#  define __LPM_qword(addr)   (* (const uint64_t*)(addr))
#  endif
#else
#define __LPM(addr)                         \
  (__extension__({                          \
      uint16_t __addr16 = (uint16_t) (addr); \
      uint8_t __result;                     \
      __LPM__1 (__result, __addr16);        \
      __result;                             \
```

```
}))

#define __LPM_word(addr)                                \
  (__extension__({                                      \
      uint16_t __addr16 = (uint16_t) (addr);            \
      uint16_t __result;                                \
      __LPM__2 (__result, __addr16);                    \
      __result;                                         \
}))

#define __LPM_dword(addr)                               \
  (__extension__({                                      \
      uint16_t __addr16 = (uint16_t) (addr);            \
      uint32_t __result;                                \
      __LPM__4 (__result, __addr16);                    \
      __result;                                         \
}))

#if __SIZEOF_LONG_LONG__ == 8
#define __LPM_qword(addr)                               \
  (__extension__({                                      \
      uint16_t __addr16 = (uint16_t) (addr);            \
      uint64_t __result;                                \
      __LPM__8 (__result, __addr16);                    \
      __result;                                         \
}))
#endif
#endif /* AVR_TINY */


#define __ELPM(addr)                                            \
  (__extension__({                                              \
      uint_farptr_t __addr32 = (addr);                          \
      uint8_t __result;                                         \
      __ELPM__1 (__result, __addr32, uint8_t);                  \
      __result;                                                 \
}))

#define __ELPM_word(addr)                                       \
  (__extension__({                                              \
      uint_farptr_t __addr32 = (addr);                          \
      uint16_t __result;                                        \
      __ELPM__2 (__result, __addr32, uint16_t);                 \
      __result;                                                 \
}))

#define __ELPM_dword(addr)                                      \
  (__extension__({                                              \
      uint_farptr_t __addr32 = (addr);                          \
      uint32_t __result;                                        \
      __ELPM__4 (__result, __addr32, uint32_t);                 \
      __result;                                                 \
}))

#if __SIZEOF_LONG_LONG__ == 8
#define __ELPM_qword(addr)                                      \
  (__extension__({                                              \
      uint_farptr_t __addr32 = (addr);                          \
      uint64_t __result;                                        \
      __ELPM__8 (__result, __addr32, uint64_t);                 \
      __result;                                                 \
}))
#endif

#endif  /* !__DOXYGEN__ */

/** \ingroup avr_pgmspace
    \def pgm_read_byte_near(__addr)
    Read a byte from the program space with a 16-bit (near) byte-address. */

#define pgm_read_byte_near(__addr) __LPM ((uint16_t)(__addr))

/** \ingroup avr_pgmspace
    \def pgm_read_word_near(__addr)
    Read a word from the program space with a 16-bit (near) byte-address. */

#define pgm_read_word_near(__addr) __LPM_word ((uint16_t)(__addr))

/** \ingroup avr_pgmspace
    \def pgm_read_dword_near(__addr)
    Read a double word from the program space with a 16-bit (near)
    byte-address.  */

#define pgm_read_dword_near(__addr) \
    __LPM_dword ((uint16_t)(__addr))

/** \ingroup avr_pgmspace
```

```
    \def pgm_read_qword_near(__addr)
    Read a quad-word from the program space with a 16-bit (near)
    byte-address.  */

#define pgm_read_qword_near(__addr) __LPM_qword ((uint16_t)(__addr))

/** \ingroup avr_pgmspace
    \def pgm_read_float_near (const float *address)
    Read a \c float from the program space with a 16-bit (near) byte-address.*/

#define pgm_read_float_near(addr)  pgm_read_float (addr)

/** \ingroup avr_pgmspace
    \def pgm_read_ptr_near(__addr)
    Read a pointer from the program space with a 16-bit (near) byte-address. */

#define pgm_read_ptr_near(__addr) \
    ((void*) __LPM_word ((uint16_t)(__addr)))

/** \ingroup avr_pgmspace
    \def pgm_read_byte_far(__addr)
    Read a byte from the program space with a 32-bit (far) byte-address. */

#define pgm_read_byte_far(__addr)  __ELPM (__addr)

/** \ingroup avr_pgmspace
    \def pgm_read_word_far(__addr)
    Read a word from the program space with a 32-bit (far) byte-address. */

#define pgm_read_word_far(__addr)  __ELPM_word (__addr)

/** \ingroup avr_pgmspace
    \def pgm_read_dword_far(__addr)
    Read a double word from the program space with a 32-bit (far)
    byte-address. */

#define pgm_read_dword_far(__addr) __ELPM_dword (__addr)

/** \ingroup avr_pgmspace
    \def pgm_read_qword_far(__addr)
    Read a quad-word from the program space with a 32-bit (far)
    byte-address. */

#define pgm_read_qword_far(__addr) __ELPM_qword (__addr)

/** \ingroup avr_pgmspace
    \def pgm_read_ptr_far(__addr)
    Read a pointer from the program space with a 32-bit (far) byte-address. */

#define pgm_read_ptr_far(__addr) ((void*) __ELPM_word (__addr))

/** \ingroup avr_pgmspace
    \def pgm_read_byte(__addr)
    Read a byte from the program space with a 16-bit (near) nyte-address. */

#define pgm_read_byte(__addr)    pgm_read_byte_near(__addr)

/** \ingroup avr_pgmspace
    \def pgm_read_word(__addr)
    Read a word from the program space with a 16-bit (near) byte-address. */

#define pgm_read_word(__addr)    pgm_read_word_near(__addr)

/** \ingroup avr_pgmspace
    \def pgm_read_dword(__addr)
    Read a double word from the program space with a 16-bit (near)
    byte-address. */

#define pgm_read_dword(__addr)   pgm_read_dword_near(__addr)

/** \ingroup avr_pgmspace
    \def pgm_read_qword(__addr)
    Read a quad-word from the program space with a 16-bit (near)
    byte-address. */

#define pgm_read_qword(__addr)   pgm_read_qword_near(__addr)

/** \ingroup avr_pgmspace
    \def pgm_read_ptr(__addr)
    Read a pointer from the program space with a 16-bit (near) byte-address. */

#define pgm_read_ptr(__addr)     pgm_read_ptr_near(__addr)

/** \ingroup avr_pgmspace
    \def pgm_get_far_address(var)

   This macro evaluates to a ::uint_farptr_t 32-bit "far" pointer (only
```

```
    24 bits used) to data even beyond the 64 KiB limit for the 16-bit ordinary
    pointer.  It is similar to the '&' operator, with some limitations.
    Example:
    \code
    #include <avr/pgmspace.h>

    // Section .progmemx.data is located after all the code sections.
    __attribute__((section(".progmemx.data")))
    const int data[] = { 2, 3, 5, 7, 9, 11 };

    int get_data (uint8_t idx)
    {
        uint_farptr_t pdata = pgm_get_far_address (data[0]);
        return pgm_read_int_far (pdata + idx * sizeof(int));
    }
    \endcode

    Comments:

    - The overhead is minimal and it's mainly due to the 32-bit size operation.

    - 24 bit sizes guarantees the code compatibility for use in future devices.

    - \p var has to be resolved at link-time as an existing symbol,
      i.e. a simple variable name, an array name, or an array or structure
      element provided the offset is known at compile-time, and \p var is
      located in static storage, etc.

    - The returned value is the symbol's \ref sec_vma "VMA"
      (virtual memory address)
      determined by the linker and falls in the corresponding memory region.
      The AVR Harvard architecture requires non-overlapping VMA areas for
      the multiple \ref sec_memory_regions "memory regions" in the processor:
      Flash ROM, RAM, and EEPROM.  Typical offset for these are
      \c 0x0, \c 0x800xx0, and \c 0x810000 respectively, derived from the
      linker script used and linker options.
*/

#define pgm_get_far_address(var)                              \
(__extension__({                                              \
    uint_farptr_t __tmp;                                      \
                                                              \
    __asm__ __volatile__ (                                    \
        "ldi    %A0, lo8(%1)"           "\n\t"                \
        "ldi    %B0, hi8(%1)"           "\n\t"                \
        "ldi    %C0, hh8(%1)"           "\n\t"                \
        "clr    %D0"                                          \
        :   "=d" (__tmp)                                      \
        :   "i"  (&(var))                                     \
    );                                                        \
    __tmp;                                                    \
}))



/** \ingroup avr_pgmspace
    \fn const void * memchr_P(const void *s, int val, size_t len)
    \brief Scan flash memory for a character.

    The memchr_P() function scans the first \p len bytes of the flash
    memory area pointed to by \p s for the character \p val.  The first
    byte to match \p val (interpreted as an unsigned character) stops
    the operation.

    \return The memchr_P() function returns a pointer to the matching
    byte or \c NULL if the character does not occur in the given memory
    area.   */
extern const void * memchr_P(const void *, int __val, size_t __len) __ATTR_CONST__;

/** \ingroup avr_pgmspace
    \fn int memcmp_P(const void *s1, const void *s2, size_t len)
    \brief Compare memory areas

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

    \returns The memcmp_P() function returns an integer less than, equal
    to, or greater than zero if the first \p len bytes of \p s1 is found,
    respectively, to be less than, to match, or be greater than the first
    \p len bytes of \p s2.  */
extern int memcmp_P(const void *, const void *, size_t) __ATTR_PURE__;

/** \ingroup avr_pgmspace
    \fn void *memccpy_P (void *dest, const void *src, int val, size_t len)

    This function is similar to memccpy() except that \p src is pointer
```

```
    to a string in program space.   */
extern void *memccpy_P(void *, const void *, int __val, size_t);

/** \ingroup avr_pgmspace
    \fn void *memcpy_P(void *dest, const void *src, size_t n)

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

    \returns The memcpy_P() function returns a pointer to dest.  */
extern void *memcpy_P(void *, const void *, size_t);

/** \ingroup avr_pgmspace
    \fn void *memmem_P(const void *s1, size_t len1, const void *s2, size_t len2)

    The memmem_P() function is similar to memmem() except that \p s2 is
    pointer to a string in program space.   */
extern void *memmem_P(const void *, size_t, const void *, size_t) __ATTR_PURE__;

/** \ingroup avr_pgmspace
    \fn const void +memrchr_P(const void *src, int val, size_t len)

    The memrchr_P() function is like the memchr_P() function, except
    that it searches backwards from the end of the \p len bytes pointed
    to by \p src instead of forwards from the front. (Glibc, GNU extension.)

    \return The memrchr_P() function returns a pointer to the matching
    byte or \c NULL if the character does not occur in the given memory
    area.   */
extern const void * memrchr_P(const void *, int __val, size_t __len) __ATTR_CONST__;

/** \ingroup avr_pgmspace
    \fn char *strcat_P(char *dest, const char *src)

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

    \returns The strcat() function returns a pointer to the resulting string
    \e dest. */
extern char *strcat_P(char *, const char *);

/** \ingroup avr_pgmspace
    \fn const char *strchr_P(const char *s, int val)
    \brief Locate character in program space string.

    The strchr_P() function locates the first occurrence of \p val
    (converted to a char) in the string pointed to by \p s in program
    space. The terminating null character is considered to be part of
    the string.

    The strchr_P() function is similar to strchr() except that \p s is
    pointer to a string in program space.

    \returns The strchr_P() function returns a pointer to the matched
    character or \c NULL if the character is not found. */
extern const char * strchr_P(const char *, int __val) __ATTR_CONST__;

/** \ingroup avr_pgmspace
    \fn const char *strchrnul_P(const char *s, int c)

    The strchrnul_P() function is like strchr_P() except that if \p c is
    not found in \p s, then it returns a pointer to the null byte at the
    end of \p s, rather than \c NULL. (Glibc, GNU extension.)

    \return The strchrnul_P() function returns a pointer to the matched
    character, or a pointer to the null byte at the end of \p s (i.e.,
    \c s+strlen(s)) if the character is not found.  */
extern const char * strchrnul_P(const char *, int __val) __ATTR_CONST__;

/** \ingroup avr_pgmspace
    \fn int strcmp_P(const char *s1, const char *s2)

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

    \returns The strcmp_P() function returns an integer less than, equal
    to, or greater than zero if \p s1 is found, respectively, to be less
    than, to match, or be greater than \p s2. A consequence of the
    ordering used by strcmp_P() is that if \p s1 is an initial substring
    of \p s2, then \p s1 is considered to be "less than" \p s2. */
extern int strcmp_P(const char *, const char *) __ATTR_PURE__;

/** \ingroup avr_pgmspace
    \fn char *strcpy_P(char *dest, const char *src)

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

```

    \returns The strcpy_P() function returns a pointer to the destination
    string dest. */
extern char *strcpy_P(char *, const char *);

/** \ingroup avr_pgmspace
    \fn int strcasecmp_P(const char *s1, const char *s2)
    \brief Compare two strings ignoring case.

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

    \param s1 A pointer to a string in the devices SRAM.
    \param s2 A pointer to a string in the devices Flash.

    \returns The strcasecmp_P() function returns an integer less than,
    equal to, or greater than zero if \p s1 is found, respectively, to
    be less than, to match, or be greater than \p s2. A consequence of
    the ordering used by strcasecmp_P() is that if \p s1 is an initial
    substring of \p s2, then \p s1 is considered to be "less than" \p s2. */
extern int strcasecmp_P(const char *, const char *) __ATTR_PURE__;

/** \ingroup avr_pgmspace
    \fn char *strcasestr_P(const char *s1, const char *s2)

    This funtion is similar to strcasestr() except that \p s2 is pointer
    to a string in program space.   */
extern char *strcasestr_P(const char *, const char *) __ATTR_PURE__;

/** \ingroup avr_pgmspace
    \fn size_t strcspn_P(const char *s, const char *reject)

    The strcspn_P() function calculates the length of the initial segment
    of \p s which consists entirely of characters not in \p reject. This
    function is similar to strcspn() except that \p reject is a pointer
    to a string in program space.

    \return The strcspn_P() function returns the number of characters in
    the initial segment of \p s which are not in the string \p reject.
    The terminating zero is not considered as a part of string. */
extern size_t strcspn_P(const char *__s, const char * __reject) __ATTR_PURE__;

/** \ingroup avr_pgmspace
    \fn size_t strlcat_P(char *dst, const char *src, size_t siz)
    \brief Concatenate two strings.

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

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

    \returns The strlcat_P() function returns strlen(src) + MIN(siz,
    strlen(initial dst)).  If retval >= siz, truncation occurred.   */
extern size_t strlcat_P (char *, const char *, size_t );

/** \ingroup avr_pgmspace
    \fn size_t strlcpy_P(char *dst, const char *src, size_t siz)
    \brief Copy a string from progmem to RAM.

    Copy \p src to string \p dst of size \p siz.  At most \p siz-1
    characters will be copied.  Always NULL terminates (unless \p siz == 0).
    The strlcpy_P() function is similar to strlcpy() except that the
    \p src is pointer to a string in memory space.

    \returns The strlcpy_P() function returns strlen(src). If
    retval >= siz, truncation occurred.  */
extern size_t strlcpy_P (char *, const char *, size_t );

/** \ingroup avr_pgmspace
    \fn size_t strnlen_P(const char *src, size_t len)
    \brief Determine the length of a fixed-size string.

    The strnlen_P() function is similar to strnlen(), except that \c src is a
    pointer to a string in program space.

    \returns The strnlen_P function returns strlen_P(src), if that is less than
    \c len, or \c len if there is no '\\0' character among the first \c len
    characters pointed to by \c src. */
extern size_t strnlen_P(const char *, size_t) __ATTR_CONST__; /* program memory can't change */

/** \ingroup avr_pgmspace
    \fn int strncmp_P(const char *s1, const char *s2, size_t n)

    The strncmp_P() function is similar to strcmp_P() except it only compares
```

```
    the first (at most) n characters of s1 and s2.

    \returns The strncmp_P() function returns an integer less than, equal to,
    or greater than zero if s1 (or the first n bytes thereof) is found,
    respectively, to be less than, to match, or be greater than s2.  */
extern int strncmp_P(const char *, const char *, size_t) __ATTR_PURE__;

/** \ingroup avr_pgmspace
    \fn int strncasecmp_P(const char *s1, const char *s2, size_t n)
    \brief Compare two strings ignoring case.

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

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

    \returns The strncasecmp_P() function returns an integer less than,
    equal to, or greater than zero if \p s1 (or the first \p n bytes
    thereof) is found, respectively, to be less than, to match, or be
    greater than \p s2. A consequence of the ordering used by
    strncasecmp_P() is that if \p s1 is an initial substring of \p s2,
    then \p s1 is considered to be "less than" \p s2.  */
extern int strncasecmp_P(const char *, const char *, size_t) __ATTR_PURE__;

/** \ingroup avr_pgmspace
    \fn char *strncat_P(char *dest, const char *src, size_t len)
    \brief Concatenate two strings.

    The strncat_P() function is similar to strncat(), except that the \e src
    string must be located in program space (flash).

    \returns The strncat_P() function returns a pointer to the resulting string
    dest.  */
extern char *strncat_P(char *, const char *, size_t);

/** \ingroup avr_pgmspace
    \fn char *strncpy_P(char *dest, const char *src, size_t n)

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

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

    \returns The strncpy_P() function returns a pointer to the destination
    string dest.  */
extern char *strncpy_P(char *, const char *, size_t);

/** \ingroup avr_pgmspace
    \fn char *strpbrk_P(const char *s, const char *accept)

    The strpbrk_P() function locates the first occurrence in the string
    \p s of any of the characters in the flash string \p accept. This
    function is similar to strpbrk() except that \p accept is a pointer
    to a string in program space.

    \return  The strpbrk_P() function returns a pointer to the character
    in \p s that matches one of the characters in \p accept, or \c NULL
    if no such character is found. The terminating zero is not considered
    as a part of string: if one or both args are empty, the result will
    \c NULL. */
extern char *strpbrk_P(const char *__s, const char * __accept) __ATTR_PURE__;

/** \ingroup avr_pgmspace
    \fn const char *strrchr_P(const char *s, int val)
    \brief Locate character in string.

    The strrchr_P() function returns a pointer to the last occurrence of
    the character \p val in the flash string \p s.

    \return The strrchr_P() function returns a pointer to the matched
    character or \c NULL if the character is not found. */
extern const char * strrchr_P(const char *, int __val) __ATTR_CONST__;

/** \ingroup avr_pgmspace
    \fn char *strsep_P(char **sp, const char *delim)
    \brief Parse a string into tokens.

    The strsep_P() function locates, in the string referenced by \p *sp,
    the first occurrence of any character in the string \p delim (or the
    terminating '\\0' character) and replaces it with a '\\0'.  The
    location of the next character after the delimiter character (or \c
    NULL, if the end of the string was reached) is stored in \p *sp. An
    "empty" field, i.e. one caused by two adjacent delimiter
```

```
    characters, can be detected by comparing the location referenced by
    the pointer returned in \p *sp to '\\0'. This function is similar to
    strsep() except that \p delim is a pointer to a string in program
    space.

    \return The strsep_P() function returns a pointer to the original
    value of \p *sp. If \p *sp is initially \c NULL, strsep_P() returns
    \c NULL. */
extern char *strsep_P(char **__sp, const char * __delim);

/** \ingroup avr_pgmspace
    \fn size_t strspn_P(const char *s, const char *accept)

    The strspn_P() function calculates the length of the initial segment
    of \p s which consists entirely of characters in \p accept. This
    function is similar to strspn() except that \p accept is a pointer
    to a string in program space.

    \return  The strspn_P() function returns the number of characters in
    the initial segment of \p s which consist only of characters from \p
    accept. The terminating zero is not considered as a part of string. */
extern size_t strspn_P(const char *__s, const char * __accept) __ATTR_PURE__;

/** \ingroup avr_pgmspace
    \fn char *strstr_P(const char *s1, const char *s2)
    \brief Locate a substring.

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

    \returns The strstr_P() function returns a pointer to the beginning
    of the substring, or NULL if the substring is not found. If \p s2
    points to a string of zero length, the function returns \p s1. */
extern char *strstr_P(const char *, const char *) __ATTR_PURE__;

/** \ingroup avr_pgmspace
    \fn char *strtok_P(char *s, const char * delim)
    \brief Parses the string into tokens.

    strtok_P() parses the string \p s into tokens. The first call to
    strtok_P() should have \p s as its first argument. Subsequent calls
    should have the first argument set to NULL. If a token ends with a
    delimiter, this delimiting character is overwritten with a '\\0' and a
    pointer to the next character is saved for the next call to strtok_P().
    The delimiter string \p delim may be different for each call.

    The strtok_P() function is similar to strtok() except that \p delim
    is pointer to a string in program space.

    \returns The strtok_P() function returns a pointer to the next token or
    NULL when no more tokens are found.

    \note strtok_P() is NOT reentrant. For a reentrant version of this
    function see strtok_rP().
 */
extern char *strtok_P(char *__s, const char * __delim);

/** \ingroup avr_pgmspace
    \fn char *strtok_rP (char *string, const char *delim, char **last)
    \brief Parses string into tokens.

    The strtok_rP() function parses \p string into tokens. The first call to
    strtok_rP() should have string as its first argument. Subsequent calls
    should have the first argument set to NULL. If a token ends with a
    delimiter, this delimiting character is overwritten with a '\\0' and a
    pointer to the next character is saved for the next call to strtok_rP().
    The delimiter string \p delim may be different for each call. \p last is
    a user allocated char* pointer. It must be the same while parsing the
    same string. strtok_rP() is a reentrant version of strtok_P().

    The strtok_rP() function is similar to strtok_r() except that \p delim
    is pointer to a string in program space.

    \returns The strtok_rP() function returns a pointer to the next token or
    NULL when no more tokens are found. */
extern char *strtok_rP(char *__s, const char * __delim, char **__last);

/** \ingroup avr_pgmspace
    \fn size_t strlen_PF(uint_farptr_t s)
    \brief Obtain the length of a string

    The strlen_PF() function is similar to strlen(), except that \e s is a
    far pointer to a string in program space.

    \param s A far pointer to the string in flash
```

```

    \returns The strlen_PF() function returns the number of characters in
    \e s. The contents of RAMPZ SFR are undefined when the function returns. */
extern size_t strlen_PF(uint_farptr_t src) __ATTR_CONST__; /* program memory can't change */

/** \ingroup avr_pgmspace
    \fn size_t strnlen_PF(uint_farptr_t s, size_t len)
    \brief Determine the length of a fixed-size string

    The strnlen_PF() function is similar to strnlen(), except that \e s is a
    far pointer to a string in program space.

    \param s A far pointer to the string in Flash
    \param len The maximum number of length to return

    \returns The strnlen_PF function returns strlen_P(\e s), if that is less
    than \e len, or \e len if there is no '\\0' character among the first \e
    len characters pointed to by \e s. The contents of RAMPZ SFR are
    undefined when the function returns. */
extern size_t strnlen_PF(uint_farptr_t src, size_t len) __ATTR_CONST__; /* program memory can't change */

/** \ingroup avr_pgmspace
    \fn void *memcpy_PF(void *dest, uint_farptr_t src, size_t n)
    \brief Copy a memory block from flash to SRAM

    The memcpy_PF() function is similar to memcpy(), except the data
    is copied from the program space and is addressed using a far pointer.

    \param dest A pointer to the destination buffer
    \param src A far pointer to the origin of data in flash memory
    \param n The number of bytes to be copied

    \returns The memcpy_PF() function returns a pointer to \e dst. The contents
    of RAMPZ SFR are undefined when the function returns. */
extern void *memcpy_PF(void *dest, uint_farptr_t src, size_t len);

/** \ingroup avr_pgmspace
    \fn char *strcpy_PF(char *dst, uint_farptr_t src)
    \brief Duplicate a string

    The strcpy_PF() function is similar to strcpy() except that \e src is a far
    pointer to a string in program space.

    \param dst A pointer to the destination string in SRAM
    \param src A far pointer to the source string in Flash

    \returns The strcpy_PF() function returns a pointer to the destination
    string \e dst. The contents of RAMPZ SFR are undefined when the funcion
    returns. */
extern char *strcpy_PF(char *dest, uint_farptr_t src);

/** \ingroup avr_pgmspace
    \fn char *strncpy_PF(char *dst, uint_farptr_t src, size_t n)
    \brief Duplicate a string until a limited length

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

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

    \param dst A pointer to the destination string in SRAM
    \param src A far pointer to the source string in Flash
    \param n The maximum number of bytes to copy

    \returns The strncpy_PF() function returns a pointer to the destination
    string \e dst. The contents of RAMPZ SFR are undefined when the function
    returns. */
extern char *strncpy_PF(char *dest, uint_farptr_t src, size_t len);

/** \ingroup avr_pgmspace
    \fn char *strcat_PF(char *dst, uint_farptr_t src)
    \brief Concatenates two strings

    The strcat_PF() function is similar to strcat() except that the \e src
    string must be located in program space (flash) and is addressed using
    a far pointer

    \param dst A pointer to the destination string in SRAM
    \param src A far pointer to the string to be appended in Flash

    \returns The strcat_PF() function returns a pointer to the resulting
    string \e dst. The contents of RAMPZ SFR are undefined when the function
    returns */
extern char *strcat_PF(char *dest, uint_farptr_t src);
```

```
/** \ingroup avr_pgmspace
    \fn size_t strlcat_PF(char *dst, uint_farptr_t src, size_t n)
    \brief Concatenate two strings

    The strlcat_PF() function is similar to strlcat(), except that the \e src
    string must be located in program space (flash) and is addressed using
    a far pointer.

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

    \param dst A pointer to the destination string in SRAM
    \param src A far pointer to the source string in Flash
    \param n The total number of bytes allocated to the destination string

    \returns The strlcat_PF() function returns strlen(\e src) + MIN(\e n,
    strlen(initial \e dst)).  If retval >= \e n, truncation occurred. The
    contents of RAMPZ SFR are undefined when the funcion returns. */
extern size_t strlcat_PF(char *dst, uint_farptr_t src, size_t siz);

/** \ingroup avr_pgmspace
    \fn char *strncat_PF(char *dst, uint_farptr_t src, size_t n)
    \brief Concatenate two strings

    The strncat_PF() function is similar to strncat(), except that the \e src
    string must be located in program space (flash) and is addressed using a
    far pointer.

    \param dst A pointer to the destination string in SRAM
    \param src A far pointer to the source string in Flash
    \param n The maximum number of bytes to append

    \returns The strncat_PF() function returns a pointer to the resulting
    string \e dst. The contents of RAMPZ SFR are undefined when the function
    returns. */
extern char *strncat_PF(char *dest, uint_farptr_t src, size_t len);

/** \ingroup avr_pgmspace
    \fn int strcmp_PF(const char *s1, uint_farptr_t s2)
    \brief Compares two strings

    The strcmp_PF() function is similar to strcmp() except that \e s2 is a far
    pointer to a string in program space.

    \param s1 A pointer to the first string in SRAM
    \param s2 A far pointer to the second string in Flash

    \returns The strcmp_PF() function returns an integer less than, equal to,
    or greater than zero if \e s1 is found, respectively, to be less than, to
    match, or be greater than \e s2. The contents of RAMPZ SFR are undefined
    when the function returns. */
extern int strcmp_PF(const char *s1, uint_farptr_t s2) __ATTR_PURE__;

/** \ingroup avr_pgmspace
    \fn int strncmp_PF(const char *s1, uint_farptr_t s2, size_t n)
    \brief Compare two strings with limited length

    The strncmp_PF() function is similar to strcmp_PF() except it only
    compares the first (at most) \e n characters of \e s1 and \e s2.

    \param s1 A pointer to the first string in SRAM
    \param s2 A far pointer to the second string in Flash
    \param n The maximum number of bytes to compare

    \returns The strncmp_PF() function returns an integer less than, equal
    to, or greater than zero if \e s1 (or the first \e n bytes thereof) is found,
    respectively, to be less than, to match, or be greater than \e s2. The
    contents of RAMPZ SFR are undefined when the function returns. */
extern int strncmp_PF(const char *s1, uint_farptr_t s2, size_t n) __ATTR_PURE__;

/** \ingroup avr_pgmspace
    \fn int strcasecmp_PF(const char *s1, uint_farptr_t s2)
    \brief Compare two strings ignoring case

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

    \param s1 A pointer to the first string in SRAM
    \param s2 A far pointer to the second string in Flash

    \returns The strcasecmp_PF() function returns an integer less than, equal
    to, or greater than zero if \e s1 is found, respectively, to be less than, to
    match, or be greater than \e s2. The contents of RAMPZ SFR are undefined
    when the function returns. */
extern int strcasecmp_PF(const char *s1, uint_farptr_t s2) __ATTR_PURE__;
```

```
/** \ingroup avr_pgmspace
    \fn int strncasecmp_PF(const char *s1, uint_farptr_t s2, size_t n)
    \brief Compare two strings ignoring case

    The strncasecmp_PF() function is similar to strcasecmp_PF(), except it
    only compares the first \e n characters of \e s1 and the string in flash is
    addressed using a far pointer.

    \param s1 A pointer to a string in SRAM
    \param s2 A far pointer to a string in Flash
    \param n The maximum number of bytes to compare

    \returns The strncasecmp_PF() function returns an integer less than, equal
    to, or greater than zero if \e s1 (or the first \e n bytes thereof) is found,
    respectively, to be less than, to match, or be greater than \e s2. The
    contents of RAMPZ SFR are undefined when the function returns.  */
extern int strncasecmp_PF(const char *s1, uint_farptr_t s2, size_t n) __ATTR_PURE__;

/** \ingroup avr_pgmspace
    \fn uint_farptr_t strchr_PF(uint_farptr_t s, int val)
    \brief Locate character in far program space string.

    The strchr_PF() function locates the first occurrence of \p val
    (converted to a char) in the string pointed to by \p s in far program
    space. The terminating null character is considered to be part of
    the string.

    The strchr_PF() function is similar to strchr() except that \p s is
    a far pointer to a string in program space that's \e not \e required to be
    located in the lower 64 KiB block like it is the case  for strchr_P().

    \returns The strchr_PF() function returns a far pointer to the matched
    character or \c 0 if the character is not found. */
extern uint_farptr_t strchr_PF(uint_farptr_t, int __val) __ATTR_CONST__;

/** \ingroup avr_pgmspace
    \fn char *strstr_PF(const char *s1, uint_farptr_t s2)
    \brief Locate a substring.

    The strstr_PF() function finds the first occurrence of the substring \c s2
    in the string \c s1.  The terminating '\\0' characters are not
    compared.
    The strstr_PF() function is similar to strstr() except that \c s2 is a
    far pointer to a string in program space.

    \returns The strstr_PF() function returns a pointer to the beginning of the
    substring, or NULL if the substring is not found.
    If \c s2 points to a string of zero length, the function returns \c s1. The
    contents of RAMPZ SFR are undefined when the function returns. */
extern char *strstr_PF(const char *s1, uint_farptr_t s2);

/** \ingroup avr_pgmspace
    \fn size_t strlcpy_PF(char *dst, uint_farptr_t src, size_t siz)
    \brief Copy a string from progmem to RAM.

    Copy src to string dst of size siz.  At most siz-1 characters will be
    copied. Always NULL terminates (unless siz == 0).

    \returns The strlcpy_PF() function returns strlen(src). If retval >= siz,
    truncation occurred.  The contents of RAMPZ SFR are undefined when the
    function returns. */
extern size_t strlcpy_PF(char *dst, uint_farptr_t src, size_t siz);

/** \ingroup avr_pgmspace
    \fn int memcmp_PF(const void *s1, uint_farptr_t s2, size_t len)
    \brief Compare memory areas

    The memcmp_PF() function compares the first \p len bytes of the memory
    areas \p s1 and flash \p s2. The comparision is performed using unsigned
    char operations. It is an equivalent of memcmp_P() function, except
    that it is capable working on all FLASH including the exteded area
    above 64kB.

    \returns The memcmp_PF() function returns an integer less than, equal
    to, or greater than zero if the first \p len bytes of \p s1 is found,
    respectively, to be less than, to match, or be greater than the first
    \p len bytes of \p s2.  */
extern int memcmp_PF(const void *, uint_farptr_t, size_t) __ATTR_PURE__;

#ifdef __DOXYGEN__
/** \ingroup avr_pgmspace
    \fn size_t strlen_P(const char *src)

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

```

```
    \returns The strlen_P() function returns the number of characters in src.

    \note strlen_P() is implemented as an inline function in the avr/pgmspace.h
    header file, which will check if the length of the string is a constant
    and known at compile time. If it is not known at compile time, the macro
    will issue a call to __strlen_P() which will then calculate the length
    of the string as normal.
*/
static inline size_t strlen_P(const char * s);
#else /* !DOXYGEN */

#ifdef __AVR_TINY__
#define __strlen_P strlen
extern size_t strlen (const char*);
#else
extern size_t __strlen_P(const char *) __ATTR_CONST__;  /* internal helper function */
#endif

static __ATTR_ALWAYS_INLINE__ size_t strlen_P(const char * s);
size_t strlen_P(const char *s)
{
  return __builtin_constant_p(__builtin_strlen(s))
     ? __builtin_strlen(s) : __strlen_P(s);
}
#endif /* DOXYGEN */

#ifdef __cplusplus
} // extern "C"
#endif

#if defined(__cplusplus) && defined(__pgm_read_template__)

/* Caveat: When this file is found via -isystem <path>, then some older
   avr-g++ versions come up with

       error: template with C linkage

   because the target description did not define NO_IMPLICIT_EXTERN_C.  */

template<typename __T, size_t>
struct __pgm_read_impl
{
  // A default implementaton for T's with a size not in { 1, 2, 3, 4, 8 }.
  // While this works, the performance is absolute scrap because GCC does
  // not handle objects well that don't fit in a register (i.e. avr-gcc
  // has no respective machine_mode).
  __T operator() (const __T *__addr) const
  {
    __T __res;
    memcpy_P (&__res, __addr, sizeof(__T));
    return __res;
  }
};

template<typename __T>
struct __pgm_read_impl<__T, 1>
{
  __T operator() (const __T *__addr) const
  {
    __T __res; __LPM__1 (__res, __addr); return __res;
  }
};

template<typename __T>
struct __pgm_read_impl<__T, 2>
{
  __T operator() (const __T *__addr) const
  {
    __T __res; __LPM__2 (__res, __addr); return __res;
  }
};

template<typename __T>
struct __pgm_read_impl<__T, 3>
{
  __T operator() (const __T *__addr) const
  {
    __T __res; __LPM__3 (__res, __addr); return __res;
  }
};

template<typename __T>
struct __pgm_read_impl<__T, 4>
{
  __T operator() (const __T *__addr) const
  {
    __T __res; __LPM__4 (__res, __addr); return __res;
```

```
  }
};

template<typename __T>
struct __pgm_read_impl<__T, 8>
{
  __T operator() (const __T *__addr) const
  {
    __T __res; __LPM__8 (__res, __addr); return __res;
  }
};

template<typename __T>
__T pgm_read (const __T *__addr)
{
  return __pgm_read_impl<__T, sizeof(__T)>() (__addr);
}

//////////////////////////////////////////////////////////

template<typename __T, size_t>
struct __pgm_read_far_impl
{
  // A default implementaton for T's with a size not in { 1, 2, 3, 4, 8 }.
  // While this works, the performance is absolute scrap because GCC does
  // not handle objects well that don't fit in a register (i.e. avr-gcc
  // has no respective machine_mode).
  __T operator() (const __T *__addr) const
  {
    __T __res;
    memcpy_PF (&__res, __addr, sizeof(__T));
    return __res;
  }
};

template<typename __T>
struct __pgm_read_far_impl<__T, 1>
{
  __T operator() (uint_farptr_t __addr) const
  {
    __T __res; __ELPM__1 (__res, __addr, __T); return __res;
  }
};

template<typename __T>
struct __pgm_read_far_impl<__T, 2>
{
  __T operator() (uint_farptr_t __addr) const
  {
    __T __res; __ELPM__2 (__res, __addr, __T); return __res;
  }
};

template<typename __T>
struct __pgm_read_far_impl<__T, 3>
{
  __T operator() (uint_farptr_t __addr) const
  {
    __T __res; __ELPM__3 (__res, __addr, __T); return __res;
  }
};

template<typename __T>
struct __pgm_read_far_impl<__T, 4>
{
  __T operator() (uint_farptr_t __addr) const
  {
    __T __res; __ELPM__4 (__res, __addr, __T); return __res;
  }
};

template<typename __T>
struct __pgm_read_far_impl<__T, 8>
{
  __T operator() (uint_farptr_t __addr) const
  {
    __T __res; __ELPM__8 (__res, __addr, __T); return __res;
  }
};

template<typename __T>
__T pgm_read_far (uint_farptr_t __addr)
{
  return __pgm_read_far_impl<__T, sizeof(__T)>() (__addr);
}

#endif /* C++ */
```

```

#ifdef __DOXYGEN__
/** \ingroup avr_pgmspace
    \fn T pgm_read<T> (const T *addr)

    Read an object of type \c T from program memory address \p addr and
    return it.
    This template is only available when macro \c __pgm_read_template__
    is defined.  */
template<typename T>
T pgm_read<T> (const T *addr);

/** \ingroup avr_pgmspace
    \fn T pgm_read_far<T> (uint_farptr_t addr)

    Read an object of type \c T from program memory address \p addr and
    return it.
    This template is only available when macro \c __pgm_read_template__
    is defined.  */
template<typename T>
T pgm_read_far<T> (uint_farptr_t addr);
#endif /* DOXYGEN */

#endif /* __PGMSPACE_H_ */
```

23.27 portpins.h

```
/* Copyright (c) 2003  Theodore A. Roth
   All rights reserved.

   Redistribution and use in source and binary forms, with or without
   modification, are permitted provided that the following conditions are met:

   * Redistributions of source code must retain the above copyright
     notice, this list of conditions and the following disclaimer.

   * Redistributions in binary form must reproduce the above copyright
     notice, this list of conditions and the following disclaimer in
     the documentation and/or other materials provided with the
     distribution.

   * Neither the name of the copyright holders nor the names of
     contributors may be used to endorse or promote products derived
     from this software without specific prior written permission.

  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
  AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
  IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
  ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
  LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
  CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
  SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
  INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
  CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
  ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
  POSSIBILITY OF SUCH DAMAGE. */

/* $Id$ */

#ifndef _AVR_PORTPINS_H_
#define _AVR_PORTPINS_H_ 1

/* This file should only be included from <avr/io.h>, never directly. */

#ifndef _AVR_IO_H_
#  error "Include <avr/io.h> instead of this file."
#endif

/* Define Generic PORTn, DDn, and PINn values. */

/* Port Data Register (generic) */
#define     PORT7        7
#define     PORT6        6
#define     PORT5        5
#define     PORT4        4
#define     PORT3        3
#define     PORT2        2
#define     PORT1        1
#define     PORT0        0

/* Port Data Direction Register (generic) */
#define     DD7          7
#define     DD6          6
#define     DD5          5
```

```
#define    DD4         4
#define    DD3         3
#define    DD2         2
#define    DD1         1
#define    DD0         0

/* Port Input Pins (generic) */
#define    PIN7        7
#define    PIN6        6
#define    PIN5        5
#define    PIN4        4
#define    PIN3        3
#define    PIN2        2
#define    PIN1        1
#define    PIN0        0

/* Define PORTxn an Pxn values for all possible port pins if not defined already by io.h. */

/* PORT A */

#if defined(PA0) && !defined(PORTA0)
#  define PORTA0 PA0
#elif defined(PORTA0) && !defined(PA0)
#  define PA0 PORTA0
#endif
#if defined(PA1) && !defined(PORTA1)
#  define PORTA1 PA1
#elif defined(PORTA1) && !defined(PA1)
#  define PA1 PORTA1
#endif
#if defined(PA2) && !defined(PORTA2)
#  define PORTA2 PA2
#elif defined(PORTA2) && !defined(PA2)
#  define PA2 PORTA2
#endif
#if defined(PA3) && !defined(PORTA3)
#  define PORTA3 PA3
#elif defined(PORTA3) && !defined(PA3)
#  define PA3 PORTA3
#endif
#if defined(PA4) && !defined(PORTA4)
#  define PORTA4 PA4
#elif defined(PORTA4) && !defined(PA4)
#  define PA4 PORTA4
#endif
#if defined(PA5) && !defined(PORTA5)
#  define PORTA5 PA5
#elif defined(PORTA5) && !defined(PA5)
#  define PA5 PORTA5
#endif
#if defined(PA6) && !defined(PORTA6)
#  define PORTA6 PA6
#elif defined(PORTA6) && !defined(PA6)
#  define PA6 PORTA6
#endif
#if defined(PA7) && !defined(PORTA7)
#  define PORTA7 PA7
#elif defined(PORTA7) && !defined(PA7)
#  define PA7 PORTA7
#endif

/* PORT B */

#if defined(PB0) && !defined(PORTB0)
#  define PORTB0 PB0
#elif defined(PORTB0) && !defined(PB0)
#  define PB0 PORTB0
#endif
#if defined(PB1) && !defined(PORTB1)
#  define PORTB1 PB1
#elif defined(PORTB1) && !defined(PB1)
#  define PB1 PORTB1
#endif
#if defined(PB2) && !defined(PORTB2)
#  define PORTB2 PB2
#elif defined(PORTB2) && !defined(PB2)
#  define PB2 PORTB2
#endif
#if defined(PB3) && !defined(PORTB3)
#  define PORTB3 PB3
#elif defined(PORTB3) && !defined(PB3)
#  define PB3 PORTB3
#endif
#if defined(PB4) && !defined(PORTB4)
#  define PORTB4 PB4
#elif defined(PORTB4) && !defined(PB4)
#  define PB4 PORTB4
```

```
#endif
#if defined(PB5) && !defined(PORTB5)
#  define PORTB5 PB5
#elif defined(PORTB5) && !defined(PB5)
#  define PB5 PORTB5
#endif
#if defined(PB6) && !defined(PORTB6)
#  define PORTB6 PB6
#elif defined(PORTB6) && !defined(PB6)
#  define PB6 PORTB6
#endif
#if defined(PB7) && !defined(PORTB7)
#  define PORTB7 PB7
#elif defined(PORTB7) && !defined(PB7)
#  define PB7 PORTB7
#endif

/* PORT C */

#if defined(PC0) && !defined(PORTC0)
#  define PORTC0 PC0
#elif defined(PORTC0) && !defined(PC0)
#  define PC0 PORTC0
#endif
#if defined(PC1) && !defined(PORTC1)
#  define PORTC1 PC1
#elif defined(PORTC1) && !defined(PC1)
#  define PC1 PORTC1
#endif
#if defined(PC2) && !defined(PORTC2)
#  define PORTC2 PC2
#elif defined(PORTC2) && !defined(PC2)
#  define PC2 PORTC2
#endif
#if defined(PC3) && !defined(PORTC3)
#  define PORTC3 PC3
#elif defined(PORTC3) && !defined(PC3)
#  define PC3 PORTC3
#endif
#if defined(PC4) && !defined(PORTC4)
#  define PORTC4 PC4
#elif defined(PORTC4) && !defined(PC4)
#  define PC4 PORTC4
#endif
#if defined(PC5) && !defined(PORTC5)
#  define PORTC5 PC5
#elif defined(PORTC5) && !defined(PC5)
#  define PC5 PORTC5
#endif
#if defined(PC6) && !defined(PORTC6)
#  define PORTC6 PC6
#elif defined(PORTC6) && !defined(PC6)
#  define PC6 PORTC6
#endif
#if defined(PC7) && !defined(PORTC7)
#  define PORTC7 PC7
#elif defined(PORTC7) && !defined(PC7)
#  define PC7 PORTC7
#endif

/* PORT D */

#if defined(PD0) && !defined(PORTD0)
#  define PORTD0 PD0
#elif defined(PORTD0) && !defined(PD0)
#  define PD0 PORTD0
#endif
#if defined(PD1) && !defined(PORTD1)
#  define PORTD1 PD1
#elif defined(PORTD1) && !defined(PD1)
#  define PD1 PORTD1
#endif
#if defined(PD2) && !defined(PORTD2)
#  define PORTD2 PD2
#elif defined(PORTD2) && !defined(PD2)
#  define PD2 PORTD2
#endif
#if defined(PD3) && !defined(PORTD3)
#  define PORTD3 PD3
#elif defined(PORTD3) && !defined(PD3)
#  define PD3 PORTD3
#endif
#if defined(PD4) && !defined(PORTD4)
#  define PORTD4 PD4
#elif defined(PORTD4) && !defined(PD4)
#  define PD4 PORTD4
#endif
```

```
#if defined(PD5) && !defined(PORTD5)
#  define PORTD5 PD5
#elif defined(PORTD5) && !defined(PD5)
#  define PD5 PORTD5
#endif
#if defined(PD6) && !defined(PORTD6)
#  define PORTD6 PD6
#elif defined(PORTD6) && !defined(PD6)
#  define PD6 PORTD6
#endif
#if defined(PD7) && !defined(PORTD7)
#  define PORTD7 PD7
#elif defined(PORTD7) && !defined(PD7)
#  define PD7 PORTD7
#endif

/* PORT E */

#if defined(PE0) && !defined(PORTE0)
#  define PORTE0 PE0
#elif defined(PORTE0) && !defined(PE0)
#  define PE0 PORTE0
#endif
#if defined(PE1) && !defined(PORTE1)
#  define PORTE1 PE1
#elif defined(PORTE1) && !defined(PE1)
#  define PE1 PORTE1
#endif
#if defined(PE2) && !defined(PORTE2)
#  define PORTE2 PE2
#elif defined(PORTE2) && !defined(PE2)
#  define PE2 PORTE2
#endif
#if defined(PE3) && !defined(PORTE3)
#  define PORTE3 PE3
#elif defined(PORTE3) && !defined(PE3)
#  define PE3 PORTE3
#endif
#if defined(PE4) && !defined(PORTE4)
#  define PORTE4 PE4
#elif defined(PORTE4) && !defined(PE4)
#  define PE4 PORTE4
#endif
#if defined(PE5) && !defined(PORTE5)
#  define PORTE5 PE5
#elif defined(PORTE5) && !defined(PE5)
#  define PE5 PORTE5
#endif
#if defined(PE6) && !defined(PORTE6)
#  define PORTE6 PE6
#elif defined(PORTE6) && !defined(PE6)
#  define PE6 PORTE6
#endif
#if defined(PE7) && !defined(PORTE7)
#  define PORTE7 PE7
#elif defined(PORTE7) && !defined(PE7)
#  define PE7 PORTE7
#endif

/* PORT F */

#if defined(PF0) && !defined(PORTF0)
#  define PORTF0 PF0
#elif defined(PORTF0) && !defined(PF0)
#  define PF0 PORTF0
#endif
#if defined(PF1) && !defined(PORTF1)
#  define PORTF1 PF1
#elif defined(PORTF1) && !defined(PF1)
#  define PF1 PORTF1
#endif
#if defined(PF2) && !defined(PORTF2)
#  define PORTF2 PF2
#elif defined(PORTF2) && !defined(PF2)
#  define PF2 PORTF2
#endif
#if defined(PF3) && !defined(PORTF3)
#  define PORTF3 PF3
#elif defined(PORTF3) && !defined(PF3)
#  define PF3 PORTF3
#endif
#if defined(PF4) && !defined(PORTF4)
#  define PORTF4 PF4
#elif defined(PORTF4) && !defined(PF4)
#  define PF4 PORTF4
#endif
#if defined(PF5) && !defined(PORTF5)
```

```
#  define PORTF5 PF5
#elif defined(PORTF5) && !defined(PF5)
#  define PF5 PORTF5
#endif
#if defined(PF6) && !defined(PORTF6)
#  define PORTF6 PF6
#elif defined(PORTF6) && !defined(PF6)
#  define PF6 PORTF6
#endif
#if defined(PF7) && !defined(PORTF7)
#  define PORTF7 PF7
#elif defined(PORTF7) && !defined(PF7)
#  define PF7 PORTF7
#endif

/* PORT G */

#if defined(PG0) && !defined(PORTG0)
#  define PORTG0 PG0
#elif defined(PORTG0) && !defined(PG0)
#  define PG0 PORTG0
#endif
#if defined(PG1) && !defined(PORTG1)
#  define PORTG1 PG1
#elif defined(PORTG1) && !defined(PG1)
#  define PG1 PORTG1
#endif
#if defined(PG2) && !defined(PORTG2)
#  define PORTG2 PG2
#elif defined(PORTG2) && !defined(PG2)
#  define PG2 PORTG2
#endif
#if defined(PG3) && !defined(PORTG3)
#  define PORTG3 PG3
#elif defined(PORTG3) && !defined(PG3)
#  define PG3 PORTG3
#endif
#if defined(PG4) && !defined(PORTG4)
#  define PORTG4 PG4
#elif defined(PORTG4) && !defined(PG4)
#  define PG4 PORTG4
#endif
#if defined(PG5) && !defined(PORTG5)
#  define PORTG5 PG5
#elif defined(PORTG5) && !defined(PG5)
#  define PG5 PORTG5
#endif
#if defined(PG6) && !defined(PORTG6)
#  define PORTG6 PG6
#elif defined(PORTG6) && !defined(PG6)
#  define PG6 PORTG6
#endif
#if defined(PG7) && !defined(PORTG7)
#  define PORTG7 PG7
#elif defined(PORTG7) && !defined(PG7)
#  define PG7 PORTG7
#endif

/* PORT H */

#if defined(PH0) && !defined(PORTH0)
#  define PORTH0 PH0
#elif defined(PORTH0) && !defined(PH0)
#  define PH0 PORTH0
#endif
#if defined(PH1) && !defined(PORTH1)
#  define PORTH1 PH1
#elif defined(PORTH1) && !defined(PH1)
#  define PH1 PORTH1
#endif
#if defined(PH2) && !defined(PORTH2)
#  define PORTH2 PH2
#elif defined(PORTH2) && !defined(PH2)
#  define PH2 PORTH2
#endif
#if defined(PH3) && !defined(PORTH3)
#  define PORTH3 PH3
#elif defined(PORTH3) && !defined(PH3)
#  define PH3 PORTH3
#endif
#if defined(PH4) && !defined(PORTH4)
#  define PORTH4 PH4
#elif defined(PORTH4) && !defined(PH4)
#  define PH4 PORTH4
#endif
#if defined(PH5) && !defined(PORTH5)
#  define PORTH5 PH5
```

```
#elif defined(PORTH5) && !defined(PH5)
#  define PH5 PORTH5
#endif
#if defined(PH6) && !defined(PORTH6)
#  define PORTH6 PH6
#elif defined(PORTH6) && !defined(PH6)
#  define PH6 PORTH6
#endif
#if defined(PH7) && !defined(PORTH7)
#  define PORTH7 PH7
#elif defined(PORTH7) && !defined(PH7)
#  define PH7 PORTH7
#endif

/* PORT J */

#if defined(PJ0) && !defined(PORTJ0)
#  define PORTJ0 PJ0
#elif defined(PORTJ0) && !defined(PJ0)
#  define PJ0 PORTJ0
#endif
#if defined(PJ1) && !defined(PORTJ1)
#  define PORTJ1 PJ1
#elif defined(PORTJ1) && !defined(PJ1)
#  define PJ1 PORTJ1
#endif
#if defined(PJ2) && !defined(PORTJ2)
#  define PORTJ2 PJ2
#elif defined(PORTJ2) && !defined(PJ2)
#  define PJ2 PORTJ2
#endif
#if defined(PJ3) && !defined(PORTJ3)
#  define PORTJ3 PJ3
#elif defined(PORTJ3) && !defined(PJ3)
#  define PJ3 PORTJ3
#endif
#if defined(PJ4) && !defined(PORTJ4)
#  define PORTJ4 PJ4
#elif defined(PORTJ4) && !defined(PJ4)
#  define PJ4 PORTJ4
#endif
#if defined(PJ5) && !defined(PORTJ5)
#  define PORTJ5 PJ5
#elif defined(PORTJ5) && !defined(PJ5)
#  define PJ5 PORTJ5
#endif
#if defined(PJ6) && !defined(PORTJ6)
#  define PORTJ6 PJ6
#elif defined(PORTJ6) && !defined(PJ6)
#  define PJ6 PORTJ6
#endif
#if defined(PJ7) && !defined(PORTJ7)
#  define PORTJ7 PJ7
#elif defined(PORTJ7) && !defined(PJ7)
#  define PJ7 PORTJ7
#endif

/* PORT K */

#if defined(PK0) && !defined(PORTK0)
#  define PORTK0 PK0
#elif defined(PORTK0) && !defined(PK0)
#  define PK0 PORTK0
#endif
#if defined(PK1) && !defined(PORTK1)
#  define PORTK1 PK1
#elif defined(PORTK1) && !defined(PK1)
#  define PK1 PORTK1
#endif
#if defined(PK2) && !defined(PORTK2)
#  define PORTK2 PK2
#elif defined(PORTK2) && !defined(PK2)
#  define PK2 PORTK2
#endif
#if defined(PK3) && !defined(PORTK3)
#  define PORTK3 PK3
#elif defined(PORTK3) && !defined(PK3)
#  define PK3 PORTK3
#endif
#if defined(PK4) && !defined(PORTK4)
#  define PORTK4 PK4
#elif defined(PORTK4) && !defined(PK4)
#  define PK4 PORTK4
#endif
#if defined(PK5) && !defined(PORTK5)
#  define PORTK5 PK5
#elif defined(PORTK5) && !defined(PK5)
```

```
#  define PK5 PORTK5
#endif
#if defined(PK6) && !defined(PORTK6)
#  define PORTK6 PK6
#elif defined(PORTK6) && !defined(PK6)
#  define PK6 PORTK6
#endif
#if defined(PK7) && !defined(PORTK7)
#  define PORTK7 PK7
#elif defined(PORTK7) && !defined(PK7)
#  define PK7 PORTK7
#endif

/* PORT L */

#if defined(PL0) && !defined(PORTL0)
#  define PORTL0 PL0
#elif defined(PORTL0) && !defined(PL0)
#  define PL0 PORTL0
#endif
#if defined(PL1) && !defined(PORTL1)
#  define PORTL1 PL1
#elif defined(PORTL1) && !defined(PL1)
#  define PL1 PORTL1
#endif
#if defined(PL2) && !defined(PORTL2)
#  define PORTL2 PL2
#elif defined(PORTL2) && !defined(PL2)
#  define PL2 PORTL2
#endif
#if defined(PL3) && !defined(PORTL3)
#  define PORTL3 PL3
#elif defined(PORTL3) && !defined(PL3)
#  define PL3 PORTL3
#endif
#if defined(PL4) && !defined(PORTL4)
#  define PORTL4 PL4
#elif defined(PORTL4) && !defined(PL4)
#  define PL4 PORTL4
#endif
#if defined(PL5) && !defined(PORTL5)
#  define PORTL5 PL5
#elif defined(PORTL5) && !defined(PL5)
#  define PL5 PORTL5
#endif
#if defined(PL6) && !defined(PORTL6)
#  define PORTL6 PL6
#elif defined(PORTL6) && !defined(PL6)
#  define PL6 PORTL6
#endif
#if defined(PL7) && !defined(PORTL7)
#  define PORTL7 PL7
#elif defined(PORTL7) && !defined(PL7)
#  define PL7 PORTL7
#endif

#endif /* _AVR_PORTPINS_H_ */
```

23.28 power.h File Reference

Macros

- #define clock_prescale_get() (clock_div_t)(CLKPR & (uint8_t)((1<<CLKPS0)|(1<<CLKPS1)|(1<<CLKPS2)|(1<<CLKPS3))

Functions

- static void power_all_enable ()
- static void power_all_disable ()
- void clock_prescale_set (clock_div_t __x)

23.29 power.h

Go to the documentation of this file.

```
/* Copyright (c) 2006, 2007, 2008  Eric B. Weddington
   Copyright (c) 2011 Frédéric Nadeau
   All rights reserved.

   Redistribution and use in source and binary forms, with or without
   modification, are permitted provided that the following conditions are met:

   * Redistributions of source code must retain the above copyright
     notice, this list of conditions and the following disclaimer.
   * Redistributions in binary form must reproduce the above copyright
     notice, this list of conditions and the following disclaimer in
     the documentation and/or other materials provided with the
     distribution.
   * Neither the name of the copyright holders nor the names of
     contributors may be used to endorse or promote products derived
     from this software without specific prior written permission.

  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
  AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
  IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
  ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
  LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
  CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
  SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
  INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
  CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
  ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
  POSSIBILITY OF SUCH DAMAGE. */

/* $Id$ */

#ifndef _AVR_POWER_H_
#define _AVR_POWER_H_   1

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

#ifndef __DOXYGEN__
#ifndef __ATTR_ALWAYS_INLINE__
#define __ATTR_ALWAYS_INLINE__ __inline__ __attribute__((__always_inline__))
#endif
#endif /* ! DOXYGEN */

/** \file */
/** \defgroup avr_power <avr/power.h>: Power Reduction Management

\code #include <avr/power.h>\endcode

Many AVRs contain a Power Reduction Register (PRR) or Registers (PRRx) that
allow you to reduce power consumption by disabling or enabling various on-board
peripherals as needed. Some devices have the XTAL Divide Control Register
(XDIV) which offer similar functionality as System Clock Prescale
Register (CLKPR).

There are many macros in this header file that provide an easy interface
to enable or disable on-board peripherals to reduce power. See the table below.

\note Not all AVR devices have a Power Reduction Register (for example
the ATmega8). On those devices without a Power Reduction Register, the
power reduction macros are not available..

\note Not all AVR devices contain the same peripherals (for example, the LCD
interface), or they will be named differently (for example, USART and
USART0). Please consult your device's datasheet, or the header file, to
find out which macros are applicable to your device.

\note For device using the XTAL Divide Control Register (XDIV), when prescaler
is used, Timer/Counter0 can only be used in asynchronous mode. Keep in mind
that Timer/Counter0 source shall be less than ¼th of peripheral clock.
Therefore, when using a typical 32.768 kHz crystal, one shall not scale
the clock below 131.072 kHz.

\anchor avr_powermacros
<small>
<table>
  <caption>Power Macros</caption>
  <tr>
    <th>Power Macro
    <th>Description
  </tr>
  <tr>
    <td>\c power_aca_disable()</td>
    <td>Disable the Analog Comparator on PortA</td>
```

```
</tr>
<tr>
  <td>\c power_aca_enable()</td>
  <td>Enable the Analog Comparator on PortA</td>
</tr>
<tr>
  <td>\c power_adc_enable()</td>
  <td>Enable the Analog to Digital Converter module</td>
</tr>
<tr>
  <td>\c power_adc_disable()</td>
  <td>Disable the Analog to Digital Converter module</td>
</tr>
<tr>
  <td>\c power_adca_disable()</td>
  <td>Disable the Analog to Digital Converter module on PortA</td>
</tr>
<tr>
  <td>\c power_adca_enable()</td>
  <td>Enable the Analog to Digital Converter module on PortA</td>
</tr>
<tr>
  <td>\c power_evsys_disable()</td>
  <td>Disable the EVSYS module</td>
</tr>
<tr>
  <td>\c power_evsys_enable()</td>
  <td>Enable the EVSYS module</td>
</tr>
<tr>
  <td>\c power_hiresc_disable()</td>
  <td>Disable the HIRES module on PortC</td>
</tr>
<tr>
  <td>\c power_hiresc_enable()</td>
  <td>Enable the HIRES module on PortC</td>
</tr>
<tr>
  <td>\c power_lcd_enable()</td>
  <td>Enable the LCD module</td>
</tr>
<tr>
  <td>\c power_lcd_disable()</td>
  <td>Disable the LCD module</td>
</tr>
<tr>
  <td>\c power_pga_enable()</td>
  <td>Enable the Programmable Gain Amplifier module</td>
</tr>
<tr>
  <td>\c power_pga_disable()</td>
  <td>Disable the Programmable Gain Amplifier module</td>
</tr>
<tr>
  <td>\c power_pscr_enable()</td>
  <td>Enable the Reduced Power Stage Controller module</td>
</tr>
<tr>
  <td>\c power_pscr_disable()</td>
  <td>Disable the Reduced Power Stage Controller module</td>
</tr>
<tr>
  <td>\c power_psc0_enable()</td>
  <td>Enable the Power Stage Controller 0 module</td>
</tr>
<tr>
  <td>\c power_psc0_disable()</td>
  <td>Disable the Power Stage Controller 0 module</td>
</tr>
<tr>
  <td>\c power_psc1_enable()</td>
  <td>Enable the Power Stage Controller 1 module</td>
</tr>
<tr>
  <td>\c power_psc1_disable()</td>
  <td>Disable the Power Stage Controller 1 module</td>
</tr>
<tr>
  <td>\c power_psc2_enable()</td>
  <td>Enable the Power Stage Controller 2 module</td>
</tr>
<tr>
  <td>\c power_psc2_disable()</td>
  <td>Disable the Power Stage Controller 2 module</td>
</tr>
<tr>
  <td>\c power_ram0_enable()</td>
```

```
    <td>Enable the SRAM block 0</td>
  </tr>
  <tr>
    <td>\c power_ram0_disable()</td>
    <td>Disable the SRAM block 0</td>
  </tr>
  <tr>
    <td>\c power_ram1_enable()</td>
    <td>Enable the SRAM block 1</td>
  </tr>
  <tr>
    <td>\c power_ram1_disable()</td>
    <td>Disable the SRAM block 1</td>
  </tr>
  <tr>
    <td>\c power_ram2_enable()</td>
    <td>Enable the SRAM block 2</td>
  </tr>
  <tr>
    <td>\c power_ram2_disable()</td>
    <td>Disable the SRAM block 2</td>
  </tr>
  <tr>
    <td>\c power_ram3_enable()</td>
    <td>Enable the SRAM block 3</td>
  </tr>
  <tr>
    <td>\c power_ram3_disable()</td>
    <td>Disable the SRAM block 3</td>
  </tr>
  <tr>
    <td>\c power_rtc_disable()</td>
    <td>Disable the RTC module</td>
  </tr>
  <tr>
    <td>\c power_rtc_enable()</td>
    <td>Enable the RTC module</td>
  </tr>
  <tr>
    <td>\c power_spi_enable()</td>
    <td>Enable the Serial Peripheral Interface module</td>
  </tr>
  <tr>
    <td>\c power_spi_disable()</td>
    <td>Disable the Serial Peripheral Interface module</td>
  </tr>
  <tr>
    <td>\c power_spic_disable()</td>
    <td>Disable the SPI module on PortC</td>
  </tr>
  <tr>
    <td>\c power_spic_enable()</td>
    <td>Enable the SPI module on PortC</td>
  </tr>
  <tr>
    <td>\c power_spid_disable()</td>
    <td>Disable the SPI module on PortD</td>
  </tr>
  <tr>
    <td>\c power_spid_enable()</td>
    <td>Enable the SPI module on PortD</td>
  </tr>
  <tr>
    <td>\c power_tc0c_disable()</td>
    <td>Disable the TC0 module on PortC</td>
  </tr>
  <tr>
    <td>\c power_tc0c_enable()</td>
    <td>Enable the TC0 module on PortC</td>
  </tr>
  <tr>
    <td>\c power_tc0d_disable()</td>
    <td>Disable the TC0 module on PortD</td>
  </tr>
  <tr>
    <td>\c power_tc0d_enable()</td>
    <td>Enable the TC0 module on PortD</td>
  </tr>
  <tr>
    <td>\c power_tc0e_disable()</td>
    <td>Disable the TC0 module on PortE</td>
  </tr>
  <tr>
    <td>\c power_tc0e_enable()</td>
    <td>Enable the TC0 module on PortE</td>
  </tr>
  <tr>
```

```
    <td>\c power_tc0f_disable()</td>
    <td>Disable the TC0 module on PortF</td>
  </tr>
  <tr>
    <td>\c power_tc0f_enable()</td>
    <td>Enable the TC0 module on PortF</td>
  </tr>
  <tr>
    <td>\c power_tc1c_disable()</td>
    <td>Disable the TC1 module on PortC</td>
  </tr>
  <tr>
    <td>\c power_tc1c_enable()</td>
    <td>Enable the TC1 module on PortC</td>
  </tr>
  <tr>
    <td>\c power_twic_disable()</td>
    <td>Disable the Two Wire Interface module on PortC</td>
  </tr>
  <tr>
    <td>\c power_twic_enable()</td>
    <td>Enable the Two Wire Interface module on PortC</td>
  </tr>
  <tr>
    <td>\c power_twie_disable()</td>
    <td>Disable the Two Wire Interface module on PortE</td>
  </tr>
  <tr>
    <td>\c power_twie_enable()</td>
    <td>Enable the Two Wire Interface module on PortE</td>
  </tr>
  <tr>
    <td>\c power_timer0_enable()</td>
    <td>Enable the Timer 0 module</td>
  </tr>
  <tr>
    <td>\c power_timer0_disable()</td>
    <td>Disable the Timer 0 module</td>
  </tr>
  <tr>
    <td>\c power_timer1_enable()</td>
    <td>Enable the Timer 1 module</td>
  </tr>
  <tr>
    <td>\c power_timer1_disable()</td>
    <td>Disable the Timer 1 module</td>
  </tr>
  <tr>
    <td>\c power_timer2_enable()</td>
    <td>Enable the Timer 2 module</td>
  </tr>
  <tr>
    <td>\c power_timer2_disable()</td>
    <td>Disable the Timer 2 module</td>
  </tr>
  <tr>
    <td>\c power_timer3_enable()</td>
    <td>Enable the Timer 3 module</td>
  </tr>
  <tr>
    <td>\c power_timer3_disable()</td>
    <td>Disable the Timer 3 module</td>
  </tr>
  <tr>
    <td>\c power_timer4_enable()</td>
    <td>Enable the Timer 4 module</td>
  </tr>
  <tr>
    <td>\c power_timer4_disable()</td>
    <td>Disable the Timer 4 module</td>
  </tr>
  <tr>
    <td>\c power_timer5_enable()</td>
    <td>Enable the Timer 5 module</td>
  </tr>
  <tr>
    <td>\c power_timer5_disable()</td>
    <td>Disable the Timer 5 module</td>
  </tr>
  <tr>
    <td>\c power_twi_enable()</td>
    <td>Enable the Two Wire Interface module</td>
  </tr>
  <tr>
    <td>\c power_twi_disable()</td>
    <td>Disable the Two Wire Interface module</td>
  </tr>
```

```
<tr>
  <td>\c power_usart_enable()</td>
  <td>Enable the USART module</td>
</tr>
<tr>
  <td>\c power_usart_disable()</td>
  <td>Disable the USART module</td>
</tr>
<tr>
  <td>\c power_usart0_enable()</td>
  <td>Enable the USART 0 module</td>
</tr>
<tr>
  <td>\c power_usart0_disable()</td>
  <td>Disable the USART 0 module</td>
</tr>
<tr>
  <td>\c power_usart1_enable()</td>
  <td>Enable the USART 1 module</td>
</tr>
<tr>
  <td>\c power_usart1_disable()</td>
  <td>Disable the USART 1 module</td>
</tr>
<tr>
  <td>\c power_usart2_enable()</td>
  <td>Enable the USART 2 module</td>
</tr>
<tr>
  <td>\c power_usart2_disable()</td>
  <td>Disable the USART 2 module</td>
</tr>
<tr>
  <td>\c power_usart3_enable()</td>
  <td>Enable the USART 3 module</td>
</tr>
<tr>
  <td>\c power_usart3_disable()</td>
  <td>Disable the USART 3 module</td>
</tr>
<tr>
  <td>\c power_usartc0_disable()</td>
  <td> Disable the USART0 module on PortC</td>
</tr>
<tr>
  <td>\c power_usartc0_enable()</td>
  <td> Enable the USART0 module on PortC</td>
</tr>
<tr>
  <td>\c power_usartd0_disable()</td>
  <td> Disable the USART0 module on PortD</td>
</tr>
<tr>
  <td>\c power_usartd0_enable()</td>
  <td> Enable the USART0 module on PortD</td>
</tr>
<tr>
  <td>\c power_usarte0_disable()</td>
  <td> Disable the USART0 module on PortE</td>
</tr>
<tr>
  <td>\c power_usarte0_enable()</td>
  <td> Enable the USART0 module on PortE</td>
</tr>
<tr>
  <td>\c power_usartf0_disable()</td>
  <td> Disable the USART0 module on PortF</td>
</tr>
<tr>
  <td>\c power_usartf0_enable()</td>
  <td> Enable the USART0 module on PortF</td>
</tr>
<tr>
  <td>\c power_usb_enable()</td>
  <td>Enable the USB module</td>
</tr>
<tr>
  <td>\c power_usb_disable()</td>
  <td>Disable the USB module</td>
</tr>
<tr>
  <td>\c power_usi_enable()</td>
  <td>Enable the Universal Serial Interface module</td>
</tr>
<tr>
  <td>\c power_usi_disable()</td>
  <td>Disable the Universal Serial Interface module</td>
```

```
  </tr>
  <tr>
    <td>\c power_vadc_enable()</td>
    <td>Enable the Voltage ADC module</td>
  </tr>
  <tr>
    <td>\c power_vadc_disable()</td>
    <td>Disable the Voltage ADC module</td>
  </tr>
  <tr>
    <td>\c power_all_enable()</td>
    <td>Enable all modules</td>
  </tr>
  <tr>
    <td>\c power_all_disable()</td>
    <td>Disable all modules</td>
  </tr>
</table>
</small>
*/

#if defined(__AVR_HAVE_PRR_PRADC)
#define power_adc_enable()      (PRR &= (uint8_t)~(1 « PRADC))
#define power_adc_disable()     (PRR |= (uint8_t)(1 « PRADC))
#endif

#if defined(__AVR_HAVE_PRR_PRCAN)
#define power_can_enable()      (PRR &= (uint8_t)~(1 « PRCAN))
#define power_can_disable()     (PRR |= (uint8_t)(1 « PRCAN))
#endif

#if defined(__AVR_HAVE_PRR_PRLCD)
#define power_lcd_enable()      (PRR &= (uint8_t)~(1 « PRLCD))
#define power_lcd_disable()     (PRR |= (uint8_t)(1 « PRLCD))
#endif

#if defined(__AVR_HAVE_PRR_PRLIN)
#define power_lin_enable()      (PRR &= (uint8_t)~(1 « PRLIN))
#define power_lin_disable()     (PRR |= (uint8_t)(1 « PRLIN))
#endif

#if defined(__AVR_HAVE_PRR_PRPSC)
#define power_psc_enable()      (PRR &= (uint8_t)~(1 « PRPSC))
#define power_psc_disable()     (PRR |= (uint8_t)(1 « PRPSC))
#endif

#if defined(__AVR_HAVE_PRR_PRPSC0)
#define power_psc0_enable()     (PRR &= (uint8_t)~(1 « PRPSC0))
#define power_psc0_disable()    (PRR |= (uint8_t)(1 « PRPSC0))
#endif

#if defined(__AVR_HAVE_PRR_PRPSC1)
#define power_psc1_enable()     (PRR &= (uint8_t)~(1 « PRPSC1))
#define power_psc1_disable()    (PRR |= (uint8_t)(1 « PRPSC1))
#endif

#if defined(__AVR_HAVE_PRR_PRPSC2)
#define power_psc2_enable()     (PRR &= (uint8_t)~(1 « PRPSC2))
#define power_psc2_disable()    (PRR |= (uint8_t)(1 « PRPSC2))
#endif

#if defined(__AVR_HAVE_PRR_PRPSCR)
#define power_pscr_enable()     (PRR &= (uint8_t)~(1 « PRPSCR))
#define power_pscr_disable()    (PRR |= (uint8_t)(1 « PRPSCR))
#endif

#if defined(__AVR_HAVE_PRR_PRSPI)
#define power_spi_enable()      (PRR &= (uint8_t)~(1 « PRSPI))
#define power_spi_disable()     (PRR |= (uint8_t)(1 « PRSPI))
#endif

#if defined(__AVR_HAVE_PRR_PRTIM0)
#define power_timer0_enable()   (PRR &= (uint8_t)~(1 « PRTIM0))
#define power_timer0_disable()  (PRR |= (uint8_t)(1 « PRTIM0))
#endif

#if defined(__AVR_HAVE_PRR_PRTIM1)
#define power_timer1_enable()   (PRR &= (uint8_t)~(1 « PRTIM1))
#define power_timer1_disable()  (PRR |= (uint8_t)(1 « PRTIM1))
#endif

#if defined(__AVR_HAVE_PRR_PRTIM2)
#define power_timer2_enable()   (PRR &= (uint8_t)~(1 « PRTIM2))
#define power_timer2_disable()  (PRR |= (uint8_t)(1 « PRTIM2))
#endif

#if defined(__AVR_HAVE_PRR_PRTWI)
```

```
#define power_twi_enable()      (PRR &= (uint8_t)~(1 « PRTWI))
#define power_twi_disable()     (PRR |= (uint8_t)(1 « PRTWI))
#endif

#if defined(__AVR_HAVE_PRR_PRUSART)
#define power_usart_enable()    (PRR &= (uint8_t)~(1 « PRUSART))
#define power_usart_disable()   (PRR |= (uint8_t)(1 « PRUSART))
#endif

#if defined(__AVR_HAVE_PRR_PRUSART0)
#define power_usart0_enable()   (PRR &= (uint8_t)~(1 « PRUSART0))
#define power_usart0_disable()  (PRR |= (uint8_t)(1 « PRUSART0))
#endif

#if defined(__AVR_HAVE_PRR_PRUSART1)
#define power_usart1_enable()   (PRR &= (uint8_t)~(1 « PRUSART1))
#define power_usart1_disable()  (PRR |= (uint8_t)(1 « PRUSART1))
#endif

#if defined(__AVR_HAVE_PRR_PRUSI)
#define power_usi_enable()      (PRR &= (uint8_t)~(1 « PRUSI))
#define power_usi_disable()     (PRR |= (uint8_t)(1 « PRUSI))
#endif

#if defined(__AVR_HAVE_PRR0_PRADC)
#define power_adc_enable()      (PRR0 &= (uint8_t)~(1 « PRADC))
#define power_adc_disable()     (PRR0 |= (uint8_t)(1 « PRADC))
#endif

#if defined(__AVR_HAVE_PRR0_PRC0)
#define power_clock_output_enable()     (PRR0 &= (uint8_t)~(1 « PRCO))
#define power_clock_output_disable()    (PRR0 |= (uint8_t)(1 « PRCO))
#endif

#if defined(__AVR_HAVE_PRR0_PRCRC)
#define power_crc_enable()              (PRR0 &= (uint8_t)~(1 « PRCRC))
#define power_crc_disable()             (PRR0 |= (uint8_t)(1 « PRCRC))
#endif

#if defined(__AVR_HAVE_PRR0_PRCU)
#define power_crypto_enable()           (PRR0 &= (uint8_t)~(1 « PRCU))
#define power_crypto_disable()          (PRR0 |= (uint8_t)(1 « PRCU))
#endif

#if defined(__AVR_HAVE_PRR0_PRDS)
#define power_irdriver_enable()         (PRR0 &= (uint8_t)~(1 « PRDS))
#define power_irdriver_disable()        (PRR0 |= (uint8_t)(1 « PRDS))
#endif

#if defined(__AVR_HAVE_PRR0_PRLFR)
#define power_lfreceiver_enable()       (PRR0 &= (uint8_t)~(1 « PRLFR))
#define power_lfreceiver_disable()      (PRR0 |= (uint8_t)(1 « PRLFR))
#endif

#if defined(__AVR_HAVE_PRR0_PRLFRS)
#define power_lfrs_enable()             (PRR0 &= (uint8_t)~(1 « PRLFRS))
#define power_lfrs_disable()            (PRR0 |= (uint8_t)(1 « PRLFRS))
#endif

#if defined(__AVR_HAVE_PRR0_PRLIN)
#define power_lin_enable()              (PRR0 &= (uint8_t)~(1 « PRLIN))
#define power_lin_disable()             (PRR0 |= (uint8_t)(1 « PRLIN))
#endif

#if defined(__AVR_HAVE_PRR0_PRPGA)
#define power_pga_enable()              (PRR0 &= (uint8_t)~(1 « PRPGA))
#define power_pga_disable()             (PRR0 |= (uint8_t)(1 « PRPGA))
#endif

#if defined(__AVR_HAVE_PRR0_PRRXDC)
#define power_receive_dsp_control_enable()  (PRR0 &= (uint8_t)~(1 « PRRXDC))
#define power_receive_dsp_control_disable() (PRR0 |= (uint8_t)(1 « PRRXDC))
#endif

#if defined(__AVR_HAVE_PRR0_PRSPI)
#define power_spi_enable()              (PRR0 &= (uint8_t)~(1 « PRSPI))
#define power_spi_disable()             (PRR0 |= (uint8_t)(1 « PRSPI))
#endif

#if defined(__AVR_HAVE_PRR0_PRT0)
#define power_timer0_enable()           (PRR0 &= (uint8_t)~(1 « PRT0))
#define power_timer0_disable()          (PRR0 |= (uint8_t)(1 « PRT0))
#endif

#if defined(__AVR_HAVE_PRR0_PRTIM0)
#define power_timer0_enable()           (PRR0 &= (uint8_t)~(1 « PRTIM0))
#define power_timer0_disable()          (PRR0 |= (uint8_t)(1 « PRTIM0))
```

```
#endif

#if defined(__AVR_HAVE_PRR0_PRT1)
#define power_timer1_enable()           (PRR0 &= (uint8_t)~(1 « PRT1))
#define power_timer1_disable()          (PRR0 |= (uint8_t)(1 « PRT1))
#endif

#if defined(__AVR_HAVE_PRR0_PRTIM1)
#define power_timer1_enable()           (PRR0 &= (uint8_t)~(1 « PRTIM1))
#define power_timer1_disable()          (PRR0 |= (uint8_t)(1 « PRTIM1))
#endif

#if defined(__AVR_HAVE_PRR0_PRT2)
#define power_timer2_enable()           (PRR0 &= (uint8_t)~(1 « PRT2))
#define power_timer2_disable()          (PRR0 |= (uint8_t)(1 « PRT2))
#endif

#if defined(__AVR_HAVE_PRR0_PRTIM2)
#define power_timer2_enable()           (PRR0 &= (uint8_t)~(1 « PRTIM2))
#define power_timer2_disable()          (PRR0 |= (uint8_t)(1 « PRTIM2))
#endif

#if defined(__AVR_HAVE_PRR0_PRT3)
#define power_timer3_enable()           (PRR0 &= (uint8_t)~(1 « PRT3))
#define power_timer3_disable()          (PRR0 |= (uint8_t)(1 « PRT3))
#endif

#if defined(__AVR_HAVE_PRR0_PRTM)
#define power_timermodulator_enable()   (PRR0 &= (uint8_t)~(1 « PRTM))
#define power_timermodulator_disable()  (PRR0 |= (uint8_t)(1 « PRTM))
#endif

#if defined(__AVR_HAVE_PRR0_PRTWI)
#define power_twi_enable()              (PRR0 &= (uint8_t)~(1 « PRTWI))
#define power_twi_disable()             (PRR0 |= (uint8_t)(1 « PRTWI))
#endif

#if defined(__AVR_HAVE_PRR0_PRTWI1)
#define power_twi1_enable()             (PRR0 &= (uint8_t)~(1 « PRTWI1))
#define power_twi1_disable()            (PRR0 |= (uint8_t)(1 « PRTWI1))
#endif

#if defined(__AVR_HAVE_PRR0_PRTXDC)
#define power_transmit_dsp_control_enable()   (PRR0 &= (uint8_t)~(1 « PRTXDC))
#define power_transmit_dsp_control_disable()  (PRR0 |= (uint8_t)(1 « PRTXDC))
#endif

#if defined(__AVR_HAVE_PRR0_PRUSART0)
#define power_usart0_enable()           (PRR0 &= (uint8_t)~(1 « PRUSART0))
#define power_usart0_disable()          (PRR0 |= (uint8_t)(1 « PRUSART0))
#endif

#if defined(__AVR_HAVE_PRR0_PRUSART1)
#define power_usart1_enable()           (PRR0 &= (uint8_t)~(1 « PRUSART1))
#define power_usart1_disable()          (PRR0 |= (uint8_t)(1 « PRUSART1))
#endif

#if defined(__AVR_HAVE_PRR0_PRVADC)
#define power_vadc_enable()             (PRR0 &= (uint8_t)~(1 « PRVADC))
#define power_vadc_disable()            (PRR0 |= (uint8_t)(1 « PRVADC))
#endif

#if defined(__AVR_HAVE_PRR0_PRVM)
#define power_voltage_monitor_enable()  (PRR0 &= (uint8_t)~(1 « PRVM))
#define power_voltage_monitor_disable() (PRR0 |= (uint8_t)(1 « PRVM))
#endif

#if defined(__AVR_HAVE_PRR0_PRVRM)
#define power_vrm_enable()              (PRR0 &= (uint8_t)~(1 « PRVRM))
#define power_vrm_disable()             (PRR0 |= (uint8_t)(1 « PRVRM))
#endif

#if defined(__AVR_HAVE_PRR1_PRAES)
#define power_aes_enable()              (PRR1 &= (uint8_t)~(1 « PRAES))
#define power_aes_disable()             (PRR1 |= (uint8_t)(1 « PRAES))
#endif

#if defined(__AVR_HAVE_PRR1_PRCI)
#define power_cinterface_enable()       (PRR1 &= (uint8_t)~(1 « PRCI))
#define power_cinterface_disable()      (PRR1 |= (uint8_t)(1 « PRCI))
#endif

#if defined(__AVR_HAVE_PRR1_PRHSSPI)
#define power_hsspi_enable()            (PRR1 &= (uint8_t)~(1 « PRHSSPI))
#define power_hsspi_disable()           (PRR1 |= (uint8_t)(1 « PRHSSPI))
#endif

```

```
#if defined(__AVR_HAVE_PRR1_PRKB)
#define power_kb_enable()               (PRR1 &= (uint8_t)~(1 « PRKB))
#define power_kb_disable()              (PRR1 |= (uint8_t)(1 « PRKB))
#endif

#if defined(__AVR_HAVE_PRR1_PRLFPH)
#define power_lfph_enable()             (PRR1 &= (uint8_t)~(1 « PRLFPH))
#define power_lfph_disable()            (PRR1 |= (uint8_t)(1 « PRLFPH))
#endif

#if defined(__AVR_HAVE_PRR1_PRLFR)
#define power_lfreceiver_enable()       (PRR1 &= (uint8_t)~(1 « PRLFR))
#define power_lfreceiver_disable()      (PRR1 |= (uint8_t)(1 « PRLFR))
#endif

#if defined(__AVR_HAVE_PRR1_PRLFTP)
#define power_lftp_enable()             (PRR1 &= (uint8_t)~(1 « PRLFTP))
#define power_lftp_disable()            (PRR1 |= (uint8_t)(1 « PRLFTP))
#endif

#if defined(__AVR_HAVE_PRR1_PRSCI)
#define power_sci_enable()              (PRR1 &= (uint8_t)~(1 « PRSCI))
#define power_sci_disable()             (PRR1 |= (uint8_t)(1 « PRSCI))
#endif

#if defined(__AVR_HAVE_PRR1_PRSPI)
#define power_spi_enable()              (PRR1 &= (uint8_t)~(1 « PRSPI))
#define power_spi_disable()             (PRR1 |= (uint8_t)(1 « PRSPI))
#endif

#if defined(__AVR_HAVE_PRR1_PRT1)
#define power_timer1_enable()           (PRR1 &= (uint8_t)~(1 « PRT1))
#define power_timer1_disable()          (PRR1 |= (uint8_t)(1 « PRT1))
#endif

#if defined(__AVR_HAVE_PRR1_PRT2)
#define power_timer2_enable()           (PRR1 &= (uint8_t)~(1 « PRT2))
#define power_timer2_disable()          (PRR1 |= (uint8_t)(1 « PRT2))
#endif

#if defined(__AVR_HAVE_PRR1_PRT3)
#define power_timer3_enable()           (PRR1 &= (uint8_t)~(1 « PRT3))
#define power_timer3_disable()          (PRR1 |= (uint8_t)(1 « PRT3))
#endif

#if defined(__AVR_HAVE_PRR1_PRT4)
#define power_timer4_enable()           (PRR1 &= (uint8_t)~(1 « PRT4))
#define power_timer4_disable()          (PRR1 |= (uint8_t)(1 « PRT4))
#endif

#if defined(__AVR_HAVE_PRR1_PRT5)
#define power_timer5_enable()           (PRR1 &= (uint8_t)~(1 « PRT5))
#define power_timer5_disable()          (PRR1 |= (uint8_t)(1 « PRT5))
#endif

#if defined(__AVR_HAVE_PRR1_PRTIM3)
#define power_timer3_enable()           (PRR1 &= (uint8_t)~(1 « PRTIM3))
#define power_timer3_disable()          (PRR1 |= (uint8_t)(1 « PRTIM3))
#endif

#if defined(__AVR_HAVE_PRR1_PRTIM4)
#define power_timer4_enable()           (PRR1 &= (uint8_t)~(1 « PRTIM4))
#define power_timer4_disable()          (PRR1 |= (uint8_t)(1 « PRTIM4))
#endif

#if defined(__AVR_HAVE_PRR1_PRTIM5)
#define power_timer5_enable()           (PRR1 &= (uint8_t)~(1 « PRTIM5))
#define power_timer5_disable()          (PRR1 |= (uint8_t)(1 « PRTIM5))
#endif

#if defined(__AVR_HAVE_PRR1_PRTRX24)
#define power_transceiver_enable()      (PRR1 &= (uint8_t)~(1 « PRTRX24))
#define power_transceiver_disable()     (PRR1 |= (uint8_t)(1 « PRTRX24))
#endif

#if defined(__AVR_HAVE_PRR1_PRUSART1)
#define power_usart1_enable()           (PRR1 &= (uint8_t)~(1 « PRUSART1))
#define power_usart1_disable()          (PRR1 |= (uint8_t)(1 « PRUSART1))
#endif

#if defined(__AVR_HAVE_PRR1_PRUSART2)
#define power_usart2_enable()           (PRR1 &= (uint8_t)~(1 « PRUSART2))
#define power_usart2_disable()          (PRR1 |= (uint8_t)(1 « PRUSART2))
#endif

#if defined(__AVR_HAVE_PRR1_PRUSART3)
#define power_usart3_enable()           (PRR1 &= (uint8_t)~(1 « PRUSART3))
```

```
#define power_usart3_disable()          (PRR1 |= (uint8_t)(1 « PRUSART3))
#endif

#if defined(__AVR_HAVE_PRR1_PRUSB)
#define power_usb_enable()              (PRR1 &= (uint8_t)~(1 « PRUSB))
#define power_usb_disable()             (PRR1 |= (uint8_t)(1 « PRUSB))
#endif

#if defined(__AVR_HAVE_PRR1_PRUSBH)
#define power_usbh_enable()             (PRR1 &= (uint8_t)~(1 « PRUSBH))
#define power_usbh_disable()            (PRR1 |= (uint8_t)(1 « PRUSBH))
#endif

#if defined(__AVR_HAVE_PRR2_PRDF)
#define power_data_fifo_enable()        (PRR2 &= (uint8_t)~(1 « PRDF))
#define power_data_fifo_disable()       (PRR2 |= (uint8_t)(1 « PRDF))
#endif

#if defined(__AVR_HAVE_PRR2_PRIDS)
#define power_id_scan_enable()          (PRR2 &= (uint8_t)~(1 « PRIDS))
#define power_id_scan_disable()         (PRR2 |= (uint8_t)(1 « PRIDS))
#endif

#if defined(__AVR_HAVE_PRR2_PRRAM0)
#define power_ram0_enable()             (PRR2 &= (uint8_t)~(1 « PRRAM0))
#define power_ram0_disable()            (PRR2 |= (uint8_t)(1 « PRRAM0))
#endif

#if defined(__AVR_HAVE_PRR2_PRRAM1)
#define power_ram1_enable()             (PRR2 &= (uint8_t)~(1 « PRRAM1))
#define power_ram1_disable()            (PRR2 |= (uint8_t)(1 « PRRAM1))
#endif

#if defined(__AVR_HAVE_PRR2_PRRAM2)
#define power_ram2_enable()             (PRR2 &= (uint8_t)~(1 « PRRAM2))
#define power_ram2_disable()            (PRR2 |= (uint8_t)(1 « PRRAM2))
#endif

#if defined(__AVR_HAVE_PRR2_PRRAM3)
#define power_ram3_enable()             (PRR2 &= (uint8_t)~(1 « PRRAM3))
#define power_ram3_disable()            (PRR2 |= (uint8_t)(1 « PRRAM3))
#endif

#if defined(__AVR_HAVE_PRR2_PRRS)
#define power_rssi_buffer_enable()      (PRR2 &= (uint8_t)~(1 « PRRS))
#define power_rssi_buffer_disable()     (PRR2 |= (uint8_t)(1 « PRRS))
#endif

#if defined(__AVR_HAVE_PRR2_PRSF)
#define power_preamble_rssi_fifo_enable()       (PRR2 &= (uint8_t)~(1 « PRSF))
#define power_preamble_rssi_fifo_disable()      (PRR2 |= (uint8_t)(1 « PRSF))
#endif

#if defined(__AVR_HAVE_PRR2_PRSPI2)
#define power_spi2_enable()             (PRR2 &= (uint8_t)~(1 « PRSPI2))
#define power_spi2_disable()            (PRR2 |= (uint8_t)(1 « PRSPI2))
#endif

#if defined(__AVR_HAVE_PRR2_PRSSM)
#define power_sequencer_state_machine_enable()  (PRR2 &= (uint8_t)~(1 « PRSSM))
#define power_sequencer_state_machine_disable() (PRR2 |= (uint8_t)(1 « PRSSM))
#endif

#if defined(__AVR_HAVE_PRR2_PRTM)
#define power_tx_modulator_enable()     (PRR2 &= (uint8_t)~(1 « PRTM))
#define power_tx_modulator_disable()    (PRR2 |= (uint8_t)(1 « PRTM))
#endif

#if defined(__AVR_HAVE_PRR2_PRTWI2)
#define power_twi2_enable()             (PRR2 &= (uint8_t)~(1 « PRTWI2))
#define power_twi2_disable()            (PRR2 |= (uint8_t)(1 « PRTWI2))
#endif

#if defined(__AVR_HAVE_PRR2_PRXA)
#define power_rx_buffer_A_enable()      (PRR2 &= (uint8_t)~(1 « PRXA))
#define power_rx_buffer_A_disable()     (PRR2 |= (uint8_t)(1 « PRXA))
#endif

#if defined(__AVR_HAVE_PRR2_PRXB)
#define power_rx_buffer_B_enable()      (PRR2 &= (uint8_t)~(1 « PRXB))
#define power_rx_buffer_B_disable()     (PRR2 |= (uint8_t)(1 « PRXB))
#endif

#if defined(__AVR_HAVE_PRGEN_AES)
#define power_aes_enable()              (PR_PRGEN &= (uint8_t)~(PR_AES_bm))
#define power_aes_disable()             (PR_PRGEN |= (uint8_t)PR_AES_bm)
#endif
```

```

#if defined(__AVR_HAVE_PRGEN_DMA)
#define power_dma_enable()              (PR_PRGEN &= (uint8_t)~(PR_DMA_bm))
#define power_dma_disable()             (PR_PRGEN |= (uint8_t)PR_DMA_bm)
#endif

#if defined(__AVR_HAVE_PRGEN_EBI)
#define power_ebi_enable()              (PR_PRGEN &= (uint8_t)~(PR_EBI_bm))
#define power_ebi_disable()             (PR_PRGEN |= (uint8_t)PR_EBI_bm)
#endif

#if defined(__AVR_HAVE_PRGEN_EDMA)
#define power_edma_enable()             (PR_PRGEN &= (uint8_t)~(PR_EDMA_bm))
#define power_edma_disable()            (PR_PRGEN |= (uint8_t)PR_EDMA_bm)
#endif

#if defined(__AVR_HAVE_PRGEN_EVSYS)
#define power_evsys_enable()            (PR_PRGEN &= (uint8_t)~(PR_EVSYS_bm))
#define power_evsys_disable()           (PR_PRGEN |= (uint8_t)PR_EVSYS_bm)
#endif

#if defined(__AVR_HAVE_PRGEN_LCD)
#define power_lcd_enable()              (PR_PRGEN &= (uint8_t)~(PR_LCD_bm))
#define power_lcd_disable()             (PR_PRGEN |= (uint8_t)PR_LCD_bm)
#endif

#if defined(__AVR_HAVE_PRGEN_RTC)
#define power_rtc_enable()              (PR_PRGEN &= (uint8_t)~(PR_RTC_bm))
#define power_rtc_disable()             (PR_PRGEN |= (uint8_t)PR_RTC_bm)
#endif

#if defined(__AVR_HAVE_PRGEN_USB)
#define power_usb_enable()              (PR_PRGEN &= (uint8_t)~(PR_USB_bm))
#define power_usb_disable()             (PR_PRGEN &= (uint8_t)(PR_USB_bm))
#endif

#if defined(__AVR_HAVE_PRGEN_XCL)
#define power_xcl_enable()              (PR_PRGEN &= (uint8_t)~(PR_XCL_bm))
#define power_xcl_disable()             (PR_PRGEN |= (uint8_t)PR_XCL_bm)
#endif

#if defined(__AVR_HAVE_PRPA_AC)
#define power_aca_enable()      (PR_PRPA &= (uint8_t)~(PR_AC_bm))
#define power_aca_disable()     (PR_PRPA |= (uint8_t)PR_AC_bm)
#endif

#if defined(__AVR_HAVE_PRPA_ADC)
#define power_adca_enable()     (PR_PRPA &= (uint8_t)~(PR_ADC_bm))
#define power_adca_disable()    (PR_PRPA |= (uint8_t)PR_ADC_bm)
#endif

#if defined(__AVR_HAVE_PRPA_DAC)
#define power_daca_enable()     (PR_PRPA &= (uint8_t)~(PR_DAC_bm))
#define power_daca_disable()    (PR_PRPA |= (uint8_t)PR_DAC_bm)
#endif

#if defined(__AVR_HAVE_PRPB_AC)
#define power_acb_enable()      (PR_PRPB &= (uint8_t)~(PR_AC_bm))
#define power_acb_disable()     (PR_PRPB |= (uint8_t)PR_AC_bm)
#endif

#if defined(__AVR_HAVE_PRPB_ADC)
#define power_adcb_enable()     (PR_PRPB &= (uint8_t)~(PR_ADC_bm))
#define power_adcb_disable()    (PR_PRPB |= (uint8_t)PR_ADC_bm)
#endif

#if defined(__AVR_HAVE_PRPB_DAC)
#define power_dacb_enable()     (PR_PRPB &= (uint8_t)~(PR_DAC_bm))
#define power_dacb_disable()    (PR_PRPB |= (uint8_t)PR_DAC_bm)
#endif

#if defined(__AVR_HAVE_PRPC_HIRES)
#define power_hiresc_enable()   (PR_PRPC &= (uint8_t)~(PR_HIRES_bm))
#define power_hiresc_disable()  (PR_PRPC |= (uint8_t)PR_HIRES_bm)
#endif

#if defined(__AVR_HAVE_PRPC_SPI)
#define power_spic_enable()     (PR_PRPC &= (uint8_t)~(PR_SPI_bm))
#define power_spic_disable()    (PR_PRPC |= (uint8_t)PR_SPI_bm)
#endif

#if defined(__AVR_HAVE_PRPC_TC0)
#define power_tc0c_enable()     (PR_PRPC &= (uint8_t)~(PR_TC0_bm))
#define power_tc0c_disable()    (PR_PRPC |= (uint8_t)PR_TC0_bm)
#endif

#if defined(__AVR_HAVE_PRPC_TC1)
```

```
#define power_tc1c_enable()      (PR_PRPC &= (uint8_t)~(PR_TC1_bm))
#define power_tc1c_disable()     (PR_PRPC |= (uint8_t)PR_TC1_bm)
#endif

#if defined(__AVR_HAVE_PRPC_TC4)
#define power_tc4c_enable()      (PR_PRPC  &= (uint8_t)~(PR_TC4_bm))
#define power_tc4c_disable()     (PR_PRPC  |= (uint8_t)PR_TC4_bm)
#endif

#if defined(__AVR_HAVE_PRPC_TC5)
#define power_tc5c_enable()      (PR_PRPC  &= (uint8_t)~(PR_TC5_bm))
#define power_tc5c_disable()     (PR_PRPC  |= (uint8_t)PR_TC5_bm)
#endif

#if defined(__AVR_HAVE_PRPC_TWI)
#define power_twic_enable()      (PR_PRPC &= (uint8_t)~(PR_TWI_bm))
#define power_twic_disable()     (PR_PRPC |= (uint8_t)PR_TWI_bm)
#endif

#if defined(__AVR_HAVE_PRPC_USART0)
#define power_usartc0_enable()   (PR_PRPC &= (uint8_t)~(PR_USART0_bm))
#define power_usartc0_disable()  (PR_PRPC |= (uint8_t)PR_USART0_bm)
#endif

#if defined(__AVR_HAVE_PRPC_USART1)
#define power_usartc1_enable()   (PR_PRPC &= (uint8_t)~(PR_USART1_bm))
#define power_usartc1_disable()  (PR_PRPC |= (uint8_t)PR_USART1_bm)
#endif

#if defined(__AVR_HAVE_PRPD_HIRES)
#define power_hiresd_enable()    (PR_PRPD &= (uint8_t)~(PR_HIRES_bm))
#define power_hiresd_disable()   (PR_PRPD |= (uint8_t)PR_HIRES_bm)
#endif

#if defined(__AVR_HAVE_PRPD_SPI)
#define power_spid_enable()      (PR_PRPD &= (uint8_t)~(PR_SPI_bm))
#define power_spid_disable()     (PR_PRPD |= (uint8_t)PR_SPI_bm)
#endif

#if defined(__AVR_HAVE_PRPD_TC0)
#define power_tc0d_enable()      (PR_PRPD &= (uint8_t)~(PR_TC0_bm))
#define power_tc0d_disable()     (PR_PRPD |= (uint8_t)PR_TC0_bm)
#endif

#if defined(__AVR_HAVE_PRPD_TC1)
#define power_tc1d_enable()      (PR_PRPD &= (uint8_t)~(PR_TC1_bm))
#define power_tc1d_disable()     (PR_PRPD |= (uint8_t)PR_TC1_bm)
#endif

#if defined(__AVR_HAVE_PRPD_TC5)
#define power_tc5d_enable()      (PR_PRPD  &= (uint8_t)~(PR_TC5_bm))
#define power_tc5d_disable()     (PR_PRPD  |= (uint8_t)PR_TC5_bm)
#endif

#if defined(__AVR_HAVE_PRPD_TWI)
#define power_twid_enable()      (PR_PRPD &= (uint8_t)~(PR_TWI_bm))
#define power_twid_disable()     (PR_PRPD |= (uint8_t)PR_TWI_bm)
#endif

#if defined(__AVR_HAVE_PRPD_USART0)
#define power_usartd0_enable()   (PR_PRPD &= (uint8_t)~(PR_USART0_bm))
#define power_usartd0_disable()  (PR_PRPD |= (uint8_t)PR_USART0_bm)
#endif

#if defined(__AVR_HAVE_PRPD_USART1)
#define power_usartd1_enable()   (PR_PRPD &= (uint8_t)~(PR_USART1_bm))
#define power_usartd1_disable()  (PR_PRPD |= (uint8_t)PR_USART1_bm)
#endif

#if defined(__AVR_HAVE_PRPE_HIRES)
#define power_hirese_enable()    (PR_PRPE &= (uint8_t)~(PR_HIRES_bm))
#define power_hirese_disable()   (PR_PRPE |= (uint8_t)PR_HIRES_bm)
#endif

#if defined(__AVR_HAVE_PRPE_SPI)
#define power_spie_enable()      (PR_PRPE &= (uint8_t)~(PR_SPI_bm))
#define power_spie_disable()     (PR_PRPE |= (uint8_t)PR_SPI_bm)
#endif

#if defined(__AVR_HAVE_PRPE_TC0)
#define power_tc0e_enable()      (PR_PRPE &= (uint8_t)~(PR_TC0_bm))
#define power_tc0e_disable()     (PR_PRPE |= (uint8_t)PR_TC0_bm)
#endif

#if defined(__AVR_HAVE_PRPE_TC1)
#define power_tc1e_enable()      (PR_PRPE &= (uint8_t)~(PR_TC1_bm))
#define power_tc1e_disable()     (PR_PRPE |= (uint8_t)PR_TC1_bm)
```

```
#endif

#if defined(__AVR_HAVE_PRPE_TWI)
#define power_twie_enable()      (PR_PRPE &= (uint8_t)~(PR_TWI_bm))
#define power_twie_disable()     (PR_PRPE |= (uint8_t)PR_TWI_bm)
#endif

#if defined(__AVR_HAVE_PRPE_USART0)
#define power_usarte0_enable()   (PR_PRPE &= (uint8_t)~(PR_USART0_bm))
#define power_usarte0_disable()  (PR_PRPE |= (uint8_t)PR_USART0_bm)
#endif

#if defined(__AVR_HAVE_PRPE_USART1)
#define power_usarte1_enable()   (PR_PRPE &= (uint8_t)~(PR_USART1_bm))
#define power_usarte1_disable()  (PR_PRPE |= (uint8_t)PR_USART1_bm)
#endif

#if defined(__AVR_HAVE_PRPF_HIRES)
#define power_hiresf_enable()    (PR_PRPF &= (uint8_t)~(PR_HIRES_bm))
#define power_hiresf_disable()   (PR_PRPF |= (uint8_t)PR_HIRES_bm)
#endif

#if defined(__AVR_HAVE_PRPF_SPI)
#define power_spif_enable()      (PR_PRPF &= (uint8_t)~(PR_SPI_bm))
#define power_spif_disable()     (PR_PRPF |= (uint8_t)PR_SPI_bm)
#endif

#if defined(__AVR_HAVE_PRPF_TC0)
#define power_tc0f_enable()      (PR_PRPF &= (uint8_t)~(PR_TC0_bm))
#define power_tc0f_disable()     (PR_PRPF |= (uint8_t)PR_TC0_bm)
#endif

#if defined(__AVR_HAVE_PRPF_TC1)
#define power_tc1f_enable()      (PR_PRPF &= (uint8_t)~(PR_TC1_bm))
#define power_tc1f_disable()     (PR_PRPF |= (uint8_t)PR_TC1_bm)
#endif

#if defined(__AVR_HAVE_PRPF_TWI)
#define power_twif_enable()      (PR_PRPF &= (uint8_t)~(PR_TWI_bm))
#define power_twif_disable()     (PR_PRPF |= (uint8_t)PR_TWI_bm)
#endif

#if defined(__AVR_HAVE_PRPF_USART0)
#define power_usartf0_enable()   (PR_PRPF &= (uint8_t)~(PR_USART0_bm))
#define power_usartf0_disable()  (PR_PRPF |= (uint8_t)PR_USART0_bm)
#endif

#if defined(__AVR_HAVE_PRPF_USART1)
#define power_usartf1_enable()   (PR_PRPF &= (uint8_t)~(PR_USART1_bm))
#define power_usartf1_disable()  (PR_PRPF |= (uint8_t)PR_USART1_bm)
#endif

#ifdef __DOXYGEN__
/**
   \ingroup avr_power
   \fn void power_all_enable()
   Enable all modules.
 */
static __ATTR_ALWAYS_INLINE__ void power_all_enable();
#else
static __ATTR_ALWAYS_INLINE__ void __power_all_enable()
{
#ifdef __AVR_HAVE_PRR
    PRR &= (uint8_t)~(__AVR_HAVE_PRR);
#endif

#ifdef __AVR_HAVE_PRR0
    PRR0 &= (uint8_t)~(__AVR_HAVE_PRR0);
#endif

#ifdef __AVR_HAVE_PRR1
    PRR1 &= (uint8_t)~(__AVR_HAVE_PRR1);
#endif

#ifdef __AVR_HAVE_PRR2
   PRR2 &= (uint8_t)~(__AVR_HAVE_PRR2);
#endif

#ifdef __AVR_HAVE_PRGEN
    PR_PRGEN &= (uint8_t)~(__AVR_HAVE_PRGEN);
#endif

#ifdef __AVR_HAVE_PRPA
    PR_PRPA &= (uint8_t)~(__AVR_HAVE_PRPA);
#endif

#ifdef __AVR_HAVE_PRPB
```

```
    PR_PRPB &= (uint8_t)~(__AVR_HAVE_PRPB);
#endif

#ifdef __AVR_HAVE_PRPC
    PR_PRPC &= (uint8_t)~(__AVR_HAVE_PRPC);
#endif

#ifdef __AVR_HAVE_PRPD
    PR_PRPD &= (uint8_t)~(__AVR_HAVE_PRPD);
#endif

#ifdef __AVR_HAVE_PRPE
    PR_PRPE &= (uint8_t)~(__AVR_HAVE_PRPE);
#endif

#ifdef __AVR_HAVE_PRPF
    PR_PRPF &= (uint8_t)~(__AVR_HAVE_PRPF);
#endif
}
#endif /* __DOXYGEN__ */

#ifdef __DOXYGEN__
/**
   \ingroup avr_power
   \fn void power_all_disable()
   Disable all modules.
 */
static __ATTR_ALWAYS_INLINE__ void power_all_disable();
#else
static __ATTR_ALWAYS_INLINE__ void __power_all_disable()
{
#ifdef __AVR_HAVE_PRR
    PRR |= (uint8_t)(__AVR_HAVE_PRR);
#endif

#ifdef __AVR_HAVE_PRR0
    PRR0 |= (uint8_t)(__AVR_HAVE_PRR0);
#endif

#ifdef __AVR_HAVE_PRR1
    PRR1 |= (uint8_t)(__AVR_HAVE_PRR1);
#endif

#ifdef __AVR_HAVE_PRR2
    PRR2 |= (uint8_t)(__AVR_HAVE_PRR2);
#endif

#ifdef __AVR_HAVE_PRGEN
    PR_PRGEN |= (uint8_t)(__AVR_HAVE_PRGEN);
#endif

#ifdef __AVR_HAVE_PRPA
    PR_PRPA |= (uint8_t)(__AVR_HAVE_PRPA);
#endif

#ifdef __AVR_HAVE_PRPB
    PR_PRPB |= (uint8_t)(__AVR_HAVE_PRPB);
#endif

#ifdef __AVR_HAVE_PRPC
    PR_PRPC |= (uint8_t)(__AVR_HAVE_PRPC);
#endif

#ifdef __AVR_HAVE_PRPD
    PR_PRPD |= (uint8_t)(__AVR_HAVE_PRPD);
#endif

#ifdef __AVR_HAVE_PRPE
    PR_PRPE |= (uint8_t)(__AVR_HAVE_PRPE);
#endif

#ifdef __AVR_HAVE_PRPF
    PR_PRPF |= (uint8_t)(__AVR_HAVE_PRPF);
#endif
}
#endif /* __DOXYGEN__ */

#ifndef __DOXYGEN__
#ifndef power_all_enable
#define power_all_enable() __power_all_enable()
#endif

#ifndef power_all_disable
#define power_all_disable() __power_all_disable()
#endif
#endif  /* !__DOXYGEN__ */
```

```
#if defined(__DOXYGEN__) \
|| defined(__AVR_AT90CAN32__) \
|| defined(__AVR_AT90CAN64__) \
|| defined(__AVR_AT90CAN128__) \
|| defined(__AVR_AT90PWM1__) \
|| defined(__AVR_AT90PWM2__) \
|| defined(__AVR_AT90PWM2B__) \
|| defined(__AVR_AT90PWM3__) \
|| defined(__AVR_AT90PWM3B__) \
|| defined(__AVR_AT90PWM81__) \
|| defined(__AVR_AT90PWM161__) \
|| defined(__AVR_AT90PWM216__) \
|| defined(__AVR_AT90PWM316__) \
|| defined(__AVR_AT90SCR100__) \
|| defined(__AVR_AT90USB646__) \
|| defined(__AVR_AT90USB647__) \
|| defined(__AVR_AT90USB82__) \
|| defined(__AVR_AT90USB1286__) \
|| defined(__AVR_AT90USB1287__) \
|| defined(__AVR_AT90USB162__) \
|| defined(__AVR_ATA5505__) \
|| defined(__AVR_ATA5272__) \
|| defined(__AVR_ATmega1280__) \
|| defined(__AVR_ATmega1281__) \
|| defined(__AVR_ATmega1284__) \
|| defined(__AVR_ATmega128RFA1__) \
|| defined(__AVR_ATmega1284RFR2__) \
|| defined(__AVR_ATmega128RFR2__) \
|| defined(__AVR_ATmega1284P__) \
|| defined(__AVR_ATmega162__) \
|| defined(__AVR_ATmega164A__) \
|| defined(__AVR_ATmega164P__) \
|| defined(__AVR_ATmega164PA__) \
|| defined(__AVR_ATmega165__) \
|| defined(__AVR_ATmega165A__) \
|| defined(__AVR_ATmega165P__) \
|| defined(__AVR_ATmega165PA__) \
|| defined(__AVR_ATmega168__) \
|| defined(__AVR_ATmega168P__) \
|| defined(__AVR_ATmega168A__) \
|| defined(__AVR_ATmega168PA__) \
|| defined(__AVR_ATmega168PB__) \
|| defined(__AVR_ATmega169__) \
|| defined(__AVR_ATmega169A__) \
|| defined(__AVR_ATmega169P__) \
|| defined(__AVR_ATmega169PA__) \
|| defined(__AVR_ATmega16M1__) \
|| defined(__AVR_ATmega16U2__) \
|| defined(__AVR_ATmega16U4__) \
|| defined(__AVR_ATmega2560__) \
|| defined(__AVR_ATmega2561__) \
|| defined(__AVR_ATmega2564RFR2__) \
|| defined(__AVR_ATmega256RFR2__) \
|| defined(__AVR_ATmega324A__) \
|| defined(__AVR_ATmega324P__) \
|| defined(__AVR_ATmega324PA__) \
|| defined(__AVR_ATmega324PB__) \
|| defined(__AVR_ATmega325__) \
|| defined(__AVR_ATmega325A__) \
|| defined(__AVR_ATmega325PA__) \
|| defined(__AVR_ATmega3250__) \
|| defined(__AVR_ATmega3250A__) \
|| defined(__AVR_ATmega3250PA__) \
|| defined(__AVR_ATmega328__) \
|| defined(__AVR_ATmega328P__) \
|| defined(__AVR_ATmega328PB__) \
|| defined(__AVR_ATmega329__) \
|| defined(__AVR_ATmega329A__) \
|| defined(__AVR_ATmega329P__) \
|| defined(__AVR_ATmega329PA__) \
|| defined(__AVR_ATmega3290__) \
|| defined(__AVR_ATmega3290A__) \
|| defined(__AVR_ATmega3290P__) \
|| defined(__AVR_ATmega3290PA__) \
|| defined(__AVR_ATmega32C1__) \
|| defined(__AVR_ATmega32M1__) \
|| defined(__AVR_ATmega32U2__) \
|| defined(__AVR_ATmega32U4__) \
|| defined(__AVR_ATmega32U6__) \
|| defined(__AVR_ATmega48__) \
|| defined(__AVR_ATmega48A__) \
|| defined(__AVR_ATmega48PA__) \
|| defined(__AVR_ATmega48P__) \
|| defined(__AVR_ATmega640__) \
|| defined(__AVR_ATmega649P__) \
|| defined(__AVR_ATmega644__) \
```

```
|| defined(__AVR_ATmega644A__) \
|| defined(__AVR_ATmega644P__) \
|| defined(__AVR_ATmega644PA__) \
|| defined(__AVR_ATmega645__) \
|| defined(__AVR_ATmega645A__) \
|| defined(__AVR_ATmega645P__) \
|| defined(__AVR_ATmega6450__) \
|| defined(__AVR_ATmega6450A__) \
|| defined(__AVR_ATmega6450P__) \
|| defined(__AVR_ATmega649__) \
|| defined(__AVR_ATmega649A__) \
|| defined(__AVR_ATmega64M1__) \
|| defined(__AVR_ATmega64C1__) \
|| defined(__AVR_ATmega6490__) \
|| defined(__AVR_ATmega6490A__) \
|| defined(__AVR_ATmega6490P__) \
|| defined(__AVR_ATmega644RFR2__) \
|| defined(__AVR_ATmega64RFR2__) \
|| defined(__AVR_ATmega88__) \
|| defined(__AVR_ATmega88A__) \
|| defined(__AVR_ATmega88P__) \
|| defined(__AVR_ATmega88PA__) \
|| defined(__AVR_ATmega8U2__) \
|| defined(__AVR_ATmega16U2__) \
|| defined(__AVR_ATmega32U2__) \
|| defined(__AVR_ATtiny48__) \
|| defined(__AVR_ATtiny88__) \
|| defined(__AVR_ATtiny87__) \
|| defined(__AVR_ATtiny167__)


/** \addtogroup avr_power

Some of the newer AVRs contain a System Clock Prescale Register (CLKPR) that
allows you to decrease the system clock frequency and the power consumption
when the need for processing power is low.
On some earlier AVRs (ATmega103, ATmega64, ATmega128), similar
functionality can be achieved through the XTAL Divide Control Register.
Below are two macros and an enumerated type that can be used to
interface to the Clock Prescale Register or
XTAL Divide Control Register.

\note Not all AVR devices have a clock prescaler. On those devices
without a Clock Prescale Register or XTAL Divide Control Register, these
macros are not available.

\code
typedef enum
{
    clock_div_1 = 0,
    clock_div_2 = 1,
    clock_div_4 = 2,
    clock_div_8 = 3,
    clock_div_16 = 4,
    clock_div_32 = 5,
    clock_div_64 = 6,
    clock_div_128 = 7,
    clock_div_256 = 8,
    clock_div_1_rc = 15, // ATmega128RFA1 only
} clock_div_t;
\endcode
Clock prescaler setting enumerations for device using
System Clock Prescale Register.

\code
typedef enum
{
    clock_div_1 = 1,
    clock_div_2 = 2,
    clock_div_4 = 4,
    clock_div_8 = 8,
    clock_div_16 = 16,
    clock_div_32 = 32,
    clock_div_64 = 64,
    clock_div_128 = 128
} clock_div_t;
\endcode
Clock prescaler setting enumerations for device using
XTAL Divide Control Register.

*/
#ifndef __DOXYGEN__
typedef enum
{
    clock_div_1 = 0,
    clock_div_2 = 1,
    clock_div_4 = 2,
```

```
    clock_div_8 = 3,
    clock_div_16 = 4,
    clock_div_32 = 5,
    clock_div_64 = 6,
    clock_div_128 = 7,
    clock_div_256 = 8
#if defined(__AVR_ATmega128RFA1__) \
|| defined(__AVR_ATmega2564RFR2__) \
|| defined(__AVR_ATmega1284RFR2__) \
|| defined(__AVR_ATmega644RFR2__) \
|| defined(__AVR_ATmega256RFR2__) \
|| defined(__AVR_ATmega128RFR2__) \
|| defined(__AVR_ATmega64RFR2__)
    , clock_div_1_rc = 15
#endif
} clock_div_t;

static __ATTR_ALWAYS_INLINE__ void clock_prescale_set(clock_div_t);
#endif  /* !__DOXYGEN__ */

/**
   \ingroup avr_power
   \fn clock_prescale_set(clock_div_t x)

Set the clock prescaler register select bits, selecting a system clock
division setting. This function is inlined, even if compiler
optimizations are disabled.

The type of \c x is \c clock_div_t.

\note For device with XTAL Divide Control Register (XDIV), \c x can actually range
from 1 to 129. Thus, one does not need to use \c clock_div_t type as argument.
*/
void clock_prescale_set(clock_div_t __x)
{
    uint8_t __tmp = _BV(CLKPCE);
    __asm__ __volatile__ (
        "in __tmp_reg__,__SREG__" "\n\t"
        "cli"                     "\n\t"
        "sts %1, %0"              "\n\t"
        "sts %1, %2"              "\n\t"
        "out __SREG__, __tmp_reg__"
        : /* no outputs */
        : "d" (__tmp),
          "M" (_SFR_MEM_ADDR(CLKPR)),
          "d" (__x)
        : "r0");
}

/** \ingroup avr_power
\def clock_prescale_get()
Gets and returns the clock prescaler register setting. The return type is \c clock_div_t.

\note For device with XTAL Divide Control Register (XDIV), return can actually
range from 1 to 129. Care should be taken has the return value could differ from the
typedef enum clock_div_t. This should only happen if clock_prescale_set was previously
called with a value other than those defined by \c clock_div_t.
*/
#define clock_prescale_get()  (clock_div_t)(CLKPR &
(uint8_t)((1«CLKPS0)|(1«CLKPS1)|(1«CLKPS2)|(1«CLKPS3)))

#elif defined(__AVR_ATmega16HVB__) \
|| defined(__AVR_ATmega16HVBREVB__) \
|| defined(__AVR_ATmega32HVB__) \
|| defined(__AVR_ATmega32HVBREVB__)

typedef enum
{
    clock_div_1 = 0,
    clock_div_2 = 1,
    clock_div_4 = 2,
    clock_div_8 = 3
} clock_div_t;

static __ATTR_ALWAYS_INLINE__ void clock_prescale_set(clock_div_t);

void clock_prescale_set(clock_div_t __x)
{
    uint8_t __tmp = _BV(CLKPCE);
    __asm__ __volatile__ (
        "in __tmp_reg__,__SREG__" "\n\t"
        "cli"                     "\n\t"
        "sts %1, %0"              "\n\t"
        "sts %1, %2"              "\n\t"
        "out __SREG__, __tmp_reg__"
        : /* no outputs */
        : "d" (__tmp),
```

```
          "M" (_SFR_MEM_ADDR(CLKPR)),
          "d" (__x)
        : "r0");
}

#define clock_prescale_get()  (clock_div_t)(CLKPR & (uint8_t)((1«CLKPS0)|(1«CLKPS1)))

#elif defined(__AVR_ATA5790__) \
|| defined (__AVR_ATA5795__)

typedef enum
{
    clock_div_1 = 0,
    clock_div_2 = 1,
    clock_div_4 = 2,
    clock_div_8 = 3,
    clock_div_16 = 4,
    clock_div_32 = 5,
    clock_div_64 = 6,
    clock_div_128 = 7,
} clock_div_t;

static __ATTR_ALWAYS_INLINE__ void system_clock_prescale_set(clock_div_t);

void system_clock_prescale_set(clock_div_t __x)
{
    uint8_t __tmp = _BV(CLKPCE);
    __asm__ __volatile__ (
        "in __tmp_reg__,__SREG__" "\n\t"
        "cli"                     "\n\t"
        "out %1, %0"              "\n\t"
        "out %1, %2"              "\n\t"
        "out __SREG__, __tmp_reg__"
        : /* no outputs */
        : "d" (__tmp),
          "I" (_SFR_IO_ADDR(CLKPR)),
          "d" (__x)
        : "r0");
}

#define system_clock_prescale_get()  (clock_div_t)(CLKPR & (uint8_t)((1«CLKPS0)|(1«CLKPS1)|(1«CLKPS2)))

typedef enum
{
    timer_clock_div_reset = 0,
    timer_clock_div_1 = 1,
    timer_clock_div_2 = 2,
    timer_clock_div_4 = 3,
    timer_clock_div_8 = 4,
    timer_clock_div_16 = 5,
    timer_clock_div_32 = 6,
    timer_clock_div_64 = 7
} timer_clock_div_t;

static __ATTR_ALWAYS_INLINE__ void timer_clock_prescale_set(timer_clock_div_t);

void timer_clock_prescale_set(timer_clock_div_t __x)
{
    uint8_t __t;
    __asm__ __volatile__ (
        "in __tmp_reg__,__SREG__"   "\n\t"
        "cli"                       "\n\t"
        "in %[temp],%[clkpr]"       "\n\t"
        "out %[clkpr],%[enable]"    "\n\t"
        "andi %[temp],%[not_CLTPS]" "\n\t"
        "or %[temp], %[set_value]"  "\n\t"
        "out %[clkpr],%[temp]"      "\n\t"
        "out __SREG__,__tmp_reg__"
        : [temp] "=d" (__t)
        : [clkpr] "I" (_SFR_IO_ADDR(CLKPR)),
          [enable] "r" (_BV(CLKPCE)),
          [not_CLTPS] "M" (0xFF & (~ ((1 « CLTPS2) | (1 « CLTPS1) | (1 « CLTPS0)))),
          [set_value] "r" ((__x & 7) « 3)
        : "r0");
}

#define timer_clock_prescale_get()  (timer_clock_div_t)(CLKPR & (uint8_t)((1«CLTPS0)|(1«CLTPS1)|(1«CLTPS2)))

#elif defined(__AVR_ATA6285__) \
|| defined(__AVR_ATA6286__)

typedef enum
{
    clock_div_1 = 0,
    clock_div_2 = 1,
```

```
    clock_div_4 = 2,
    clock_div_8 = 3,
    clock_div_16 = 4,
    clock_div_32 = 5,
    clock_div_64 = 6,
    clock_div_128 = 7
} clock_div_t;

static __ATTR_ALWAYS_INLINE__ void system_clock_prescale_set(clock_div_t);

void system_clock_prescale_set(clock_div_t __x)
{
    uint8_t __t;
    __asm__ __volatile__ (
        "in __tmp_reg__,__SREG__"   "\n\t"
        "cli"                       "\n\t"
        "in %[temp],%[clpr]"        "\n\t"
        "out %[clpr],%[enable]"     "\n\t"
        "andi %[temp],%[not_CLKPS]" "\n\t"
        "or %[temp], %[set_value]"  "\n\t"
        "out %[clpr],%[temp]"       "\n\t"
        "out __SREG__,__tmp_reg__"
        : [temp] "=d" (__t)
        : [clpr] "I" (_SFR_IO_ADDR(CLKPR)),
          [enable] "r" _BV(CLPCE),
          [not_CLKPS] "M" (0xFF & (~ ((1 « CLKPS2) | (1 « CLKPS1) | (1 « CLKPS0)))),
          [set_value] "r" (__x & 7)
        : "r0");
}

#define system_clock_prescale_get()  (clock_div_t)(CLKPR & (uint8_t)((1«CLKPS0)|(1«CLKPS1)|(1«CLKPS2)))

typedef enum
{
    timer_clock_div_reset = 0,
    timer_clock_div_1 = 1,
    timer_clock_div_2 = 2,
    timer_clock_div_4 = 3,
    timer_clock_div_8 = 4,
    timer_clock_div_16 = 5,
    timer_clock_div_32 = 6,
    timer_clock_div_64 = 7
} timer_clock_div_t;

static __ATTR_ALWAYS_INLINE__ void timer_clock_prescale_set(timer_clock_div_t);

void timer_clock_prescale_set(timer_clock_div_t __x)
{
    uint8_t __t;
    __asm__ __volatile__ (
        "in __tmp_reg__,__SREG__"   "\n\t"
        "cli"                       "\n\t"
        "in %[temp],%[clpr]"        "\n\t"
        "out %[clpr],%[enable]"     "\n\t"
        "andi %[temp],%[not_CLTPS]" "\n\t"
        "or %[temp], %[set_value]"  "\n\t"
        "out %[clpr],%[temp]"       "\n\t"
        "out __SREG__,__tmp_reg__"
        : [temp] "=d" (__t)
        : [clpr] "I" (_SFR_IO_ADDR(CLKPR)),
          [enable] "r" (_BV(CLPCE)),
          [not_CLTPS] "M" (0xFF & (~ ((1 « CLTPS2) | (1 « CLTPS1) | (1 « CLTPS0)))),
          [set_value] "r" ((__x & 7) « 3)
        : "r0");
}

#define timer_clock_prescale_get()  (timer_clock_div_t)(CLKPR & (uint8_t)((1«CLTPS0)|(1«CLTPS1)|(1«CLTPS2)))

#elif defined(__AVR_ATtiny24__) \
|| defined(__AVR_ATtiny24A__) \
|| defined(__AVR_ATtiny44__) \
|| defined(__AVR_ATtiny44A__) \
|| defined(__AVR_ATtiny84__) \
|| defined(__AVR_ATtiny84A__) \
|| defined(__AVR_ATtiny25__) \
|| defined(__AVR_ATtiny45__) \
|| defined(__AVR_ATtiny85__) \
|| defined(__AVR_ATtiny261A__) \
|| defined(__AVR_ATtiny261__) \
|| defined(__AVR_ATtiny461__) \
|| defined(__AVR_ATtiny461A__) \
|| defined(__AVR_ATtiny861__) \
|| defined(__AVR_ATtiny861A__) \
|| defined(__AVR_ATtiny2313__) \
|| defined(__AVR_ATtiny2313A__) \
```

```
|| defined(__AVR_ATtiny4313__) \
|| defined(__AVR_ATtiny13__) \
|| defined(__AVR_ATtiny13A__) \
|| defined(__AVR_ATtiny43U__) \

typedef enum
{
    clock_div_1 = 0,
    clock_div_2 = 1,
    clock_div_4 = 2,
    clock_div_8 = 3,
    clock_div_16 = 4,
    clock_div_32 = 5,
    clock_div_64 = 6,
    clock_div_128 = 7,
    clock_div_256 = 8
} clock_div_t;

static __ATTR_ALWAYS_INLINE__ void clock_prescale_set(clock_div_t);

void clock_prescale_set(clock_div_t __x)
{
    uint8_t __tmp = _BV(CLKPCE);
    __asm__ __volatile__ (
        "in __tmp_reg__,__SREG__" "\n\t"
        "cli"                     "\n\t"
        "out %1, %0"              "\n\t"
        "out %1, %2"              "\n\t"
        "out __SREG__, __tmp_reg__"
        : /* no outputs */
        : "d" (__tmp),
          "I" (_SFR_IO_ADDR(CLKPR)),
          "d" (__x)
        : "r0");
}


#define clock_prescale_get()  (clock_div_t)(CLKPR &
(uint8_t)((1«CLKPS0)|(1«CLKPS1)|(1«CLKPS2)|(1«CLKPS3)))

#elif defined(__AVR_ATtiny441__) \
|| defined(__AVR_ATtiny841__)

typedef enum
{
    clock_div_1 = 0,
    clock_div_2 = 1,
    clock_div_4 = 2,
    clock_div_8 = 3,
    clock_div_16 = 4,
    clock_div_32 = 5,
    clock_div_64 = 6,
    clock_div_128 = 7,
    clock_div_256 = 8
} clock_div_t;

static __ATTR_ALWAYS_INLINE__ void clock_prescale_set (clock_div_t);

void clock_prescale_set (clock_div_t __x)
{
    __asm__ __volatile__ (
        "in __tmp_reg__,__SREG__" "\n\t"
        "cli"                     "\n\t"
        "sts %2, %3"              "\n\t"
        "sts %1, %0"              "\n\t"
        "out __SREG__, __tmp_reg__"
        : /* no outputs */
        : "r" (__x),
          "n" (_SFR_MEM_ADDR(CLKPR)),
          "n" (_SFR_MEM_ADDR(CCP)),
          "r" ((uint8_t) 0xD8)
        : "r0");
}

#define clock_prescale_get()  (clock_div_t) (CLKPR &
(uint8_t)((1«CLKPS0)|(1«CLKPS1)|(1«CLKPS2)|(1«CLKPS3)))

#elif defined(__AVR_ATmega64__) \
|| defined(__AVR_ATmega103__) \
|| defined(__AVR_ATmega128__)

//Enum is declared for code compatibility
typedef enum
{
    clock_div_1 = 1,
    clock_div_2 = 2,
    clock_div_4 = 4,
```

```
    clock_div_8 = 8,
    clock_div_16 = 16,
    clock_div_32 = 32,
    clock_div_64 = 64,
    clock_div_128 = 128
} clock_div_t;

static __ATTR_ALWAYS_INLINE__ void clock_prescale_set(clock_div_t);

void clock_prescale_set(clock_div_t __x)
{
    if((__x <= 0) || (__x > 129))
    {
        return;//Invalid value.
    }
    else
    {
        uint8_t __tmp = 0;
        //Algo explained:
        //1 - Clear XDIV in order for it to accept a new value (actually only
        //    XDIVEN need to be cleared, but clearing XDIV is faster than
        //    read-modify-write since we will rewrite XDIV later anyway)
        //2 - wait 8 clock cycle for stability, see datasheet errata
        //3 - Exit if requested prescaler is 1
        //4 - Calculate XDIV6..0 value = 129 - __x
        //5 - Set XDIVEN bit in calculated value
        //6 - write XDIV with calculated value
        //7 - wait 8 clock cycle for stability, see datasheet errata
        __asm__ __volatile__ (
            "in __tmp_reg__,__SREG__" "\n\t"
            "cli"                     "\n\t"
            "out %2, __zero_reg__"    "\n\t"
            "nop" "\n\t"
            "nop" "\n\t"
            "nop" "\n\t"
            "nop" "\n\t"
            "nop" "\n\t"
            "nop" "\n\t"
            "nop" "\n\t"
            "nop" "\n\t"
            "cpi %1, 0x01" "\n\t"
            "breq L_%="    "\n\t"
            "ldi %0, 0x81" "\n\t" //129
            "sub %0, %1"   "\n\t"
            "ori %0, 0x80" "\n\t" //128
            "out %2, %0"   "\n\t"
            "nop" "\n\t"
            "nop" "\n\t"
            "nop" "\n\t"
            "nop" "\n\t"
            "nop" "\n\t"
            "nop" "\n\t"
            "nop" "\n\t"
            "nop" "\n\t"
            "L_%=: " "out __SREG__, __tmp_reg__"
            : "=d" (__tmp)
            : "d" (__x),
              "I" (_SFR_IO_ADDR(XDIV))
            : "r0");
    }
}

static __ATTR_ALWAYS_INLINE__ clock_div_t clock_prescale_get(void);

clock_div_t clock_prescale_get(void)
{
    if (bit_is_clear(XDIV, XDIVEN))
    {
        return 1;
    }
    else
    {
        return (clock_div_t) (129 - (XDIV & 0x7F));
    }
}

#elif defined(__AVR_ATtiny4__) \
|| defined(__AVR_ATtiny5__) \
|| defined(__AVR_ATtiny9__) \
|| defined(__AVR_ATtiny10__) \
|| defined(__AVR_ATtiny102__) \
|| defined(__AVR_ATtiny104__) \
|| defined(__AVR_ATtiny20__) \
|| defined(__AVR_ATtiny40__) \

typedef enum
{
```

```
    clock_div_1 = 0,
    clock_div_2 = 1,
    clock_div_4 = 2,
    clock_div_8 = 3,
    clock_div_16 = 4,
    clock_div_32 = 5,
    clock_div_64 = 6,
    clock_div_128 = 7,
    clock_div_256 = 8
} clock_div_t;

static __ATTR_ALWAYS_INLINE__ void clock_prescale_set(clock_div_t);

void clock_prescale_set(clock_div_t __x)
{
    uint8_t __tmp = 0xD8;
    __asm__ __volatile__ (
        "in __tmp_reg__,__SREG__" "\n\t"
        "cli"                     "\n\t"
        "out %1, %0"              "\n\t"
        "out %2, %3"              "\n\t"
        "out __SREG__, __tmp_reg__"
        : /* no outputs */
        : "d" (__tmp),
          "I" (_SFR_IO_ADDR(CCP)),
          "I" (_SFR_IO_ADDR(CLKPSR)),
          "d" (__x)
        : "r16");
}

#define clock_prescale_get()  (clock_div_t)(CLKPSR & (uint8_t)((1«CLKPS0)|(1«CLKPS1)|(1«CLKPS2)|(1«CLKPS3)))

#endif

#endif /* _AVR_POWER_H_ */
```

23.30 sfr_defs.h

```
/* Copyright (c) 2002, Marek Michalkiewicz <marekm@amelek.gda.pl>
   All rights reserved.

   Redistribution and use in source and binary forms, with or without
   modification, are permitted provided that the following conditions are met:

   * Redistributions of source code must retain the above copyright
     notice, this list of conditions and the following disclaimer.

   * Redistributions in binary form must reproduce the above copyright
     notice, this list of conditions and the following disclaimer in
     the documentation and/or other materials provided with the
     distribution.

   * Neither the name of the copyright holders nor the names of
     contributors may be used to endorse or promote products derived
     from this software without specific prior written permission.

   THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
   AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
   IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
   ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
   LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
   CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
   SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
   INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
   CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
   ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
   POSSIBILITY OF SUCH DAMAGE.  */

/* avr/sfr_defs.h - macros for accessing AVR special function registers */

/* $Id$ */

#ifndef _AVR_SFR_DEFS_H_
#define _AVR_SFR_DEFS_H_ 1

/** \defgroup avr_sfr_notes Additional notes from <avr/sfr_defs.h>
    \ingroup avr_sfr

   The \c <avr/sfr_defs.h> file is included by all of the \c <avr/ioXXXX.h>
   files, which use macros defined here to make the special function register
   definitions look like C variables or simple constants, depending on the
   <tt>_SFR_ASM_COMPAT</tt> define.  Some examples from \c <avr/iocanxx.h> to
   show how to define such macros:
```

```
\code
#define PORTA   _SFR_IO8(0x02)
#define EEAR    _SFR_IO16(0x21)
#define UDR0    _SFR_MEM8(0xC6)
#define TCNT3   _SFR_MEM16(0x94)
#define CANIDT  _SFR_MEM32(0xF0)
\endcode

   If \c _SFR_ASM_COMPAT is not defined, C programs can use names like
   <tt>PORTA</tt> directly in C expressions (also on the left side of
   assignment operators) and GCC will do the right thing (use short I/O
   instructions if possible).  The \c __SFR_OFFSET definition is not used in
   any way in this case.

   Define \c _SFR_ASM_COMPAT as 1 to make these names work as simple constants
   (addresses of the I/O registers).  This is necessary when included in
   preprocessed assembler (*.S) source files, so it is done automatically if
   \c __ASSEMBLER__ is defined.  By default, all addresses are defined as if
   they were memory addresses (used in \c lds/sts instructions).  To use these
   addresses in \c in/out instructions, you must subtract 0x20 from them.

   For more backwards compatibility, insert the following at the start of your
   old assembler source file:

\code
#define __SFR_OFFSET 0
\endcode

   This automatically subtracts 0x20 from I/O space addresses, but it's a
   hack, so it is recommended to change your source: wrap such addresses in
   macros defined here, as shown below.  After this is done, the
   <tt>__SFR_OFFSET</tt> definition is no longer necessary and can be removed.

   Real example - this code could be used in a boot loader that is portable
   between devices with \c SPMCR at different addresses.

\verbatim
<avr/iom163.h>: #define SPMCR _SFR_IO8(0x37)
<avr/iom128.h>: #define SPMCR _SFR_MEM8(0x68)
\endverbatim

\code
#if _SFR_IO_REG_P(SPMCR)
    out _SFR_IO_ADDR(SPMCR), r24
#else
    sts _SFR_MEM_ADDR(SPMCR), r24
#endif
\endcode

   You can use the \c in/out/cbi/sbi/sbic/sbis instructions, without the
   <tt>_SFR_IO_REG_P</tt> test, if you know that the register is in the I/O
   space (as with \c SREG, for example).  If it isn't, the assembler will
   complain (I/O address out of range 0...0x3f), so this should be fairly
   safe.

   If you do not define \c __SFR_OFFSET (so it will be 0x20 by default), all
   special register addresses are defined as memory addresses (so \c SREG is
   0x5f), and (if code size and speed are not important, and you don't like
   the ugly \#if above) you can always use lds/sts to access them.  But, this
   will not work if <tt>__SFR_OFFSET</tt> != 0x20, so use a different macro
   (defined only if <tt>__SFR_OFFSET</tt> == 0x20) for safety:

\code
    sts _SFR_ADDR(SPMCR), r24
\endcode

   In C programs, all 3 combinations of \c _SFR_ASM_COMPAT and
   <tt>__SFR_OFFSET</tt> are supported - the \c _SFR_ADDR(SPMCR) macro can be
   used to get the address of the \c SPMCR register (0x57 or 0x68 depending on
   device). */

#ifdef __ASSEMBLER__
#define _SFR_ASM_COMPAT 1
#elif !defined(_SFR_ASM_COMPAT)
#define _SFR_ASM_COMPAT 0
#endif

#ifndef __ASSEMBLER__
/* These only work in C programs.  */
#include <inttypes.h>

#define _MMIO_BYTE(mem_addr) (*(volatile uint8_t *)(mem_addr))
#define _MMIO_WORD(mem_addr) (*(volatile uint16_t *)(mem_addr))
#define _MMIO_DWORD(mem_addr) (*(volatile uint32_t *)(mem_addr))
#endif
```

```
#if _SFR_ASM_COMPAT

#ifndef __SFR_OFFSET
/* Define as 0 before including this file for compatibility with old asm
   sources that don't subtract __SFR_OFFSET from symbolic I/O addresses.  */
#  if __AVR_ARCH__ >= 100
#    define __SFR_OFFSET 0x00
#  else
#    define __SFR_OFFSET 0x20
#  endif
#endif

#if (__SFR_OFFSET != 0) && (__SFR_OFFSET != 0x20)
#error "__SFR_OFFSET must be 0 or 0x20"
#endif

#define _SFR_MEM8(mem_addr) (mem_addr)
#define _SFR_MEM16(mem_addr) (mem_addr)
#define _SFR_MEM32(mem_addr) (mem_addr)
#define _SFR_IO8(io_addr) ((io_addr) + __SFR_OFFSET)
#define _SFR_IO16(io_addr) ((io_addr) + __SFR_OFFSET)

#define _SFR_IO_ADDR(sfr) ((sfr) - __SFR_OFFSET)
#define _SFR_MEM_ADDR(sfr) (sfr)
#define _SFR_IO_REG_P(sfr) ((sfr) < 0x40 + __SFR_OFFSET)

#if (__SFR_OFFSET == 0x20)
/* No need to use ?: operator, so works in assembler too.  */
#define _SFR_ADDR(sfr) _SFR_MEM_ADDR(sfr)
#elif !defined(__ASSEMBLER__)
#define _SFR_ADDR(sfr) (_SFR_IO_REG_P(sfr) ? (_SFR_IO_ADDR(sfr) + 0x20) : _SFR_MEM_ADDR(sfr))
#endif

#else  /* !_SFR_ASM_COMPAT */

#ifndef __SFR_OFFSET
#  if __AVR_ARCH__ >= 100
#    define __SFR_OFFSET 0x00
#  else
#    define __SFR_OFFSET 0x20
#  endif
#endif

#define _SFR_MEM8(mem_addr) _MMIO_BYTE(mem_addr)
#define _SFR_MEM16(mem_addr) _MMIO_WORD(mem_addr)
#define _SFR_MEM32(mem_addr) _MMIO_DWORD(mem_addr)
#define _SFR_IO8(io_addr) _MMIO_BYTE((io_addr) + __SFR_OFFSET)
#define _SFR_IO16(io_addr) _MMIO_WORD((io_addr) + __SFR_OFFSET)

#define _SFR_MEM_ADDR(sfr) ((uint16_t) &(sfr))
#define _SFR_IO_ADDR(sfr) (_SFR_MEM_ADDR(sfr) - __SFR_OFFSET)
#define _SFR_IO_REG_P(sfr) (_SFR_MEM_ADDR(sfr) < 0x40 + __SFR_OFFSET)

#define _SFR_ADDR(sfr) _SFR_MEM_ADDR(sfr)

#endif /* !_SFR_ASM_COMPAT */

#define _SFR_BYTE(sfr) _MMIO_BYTE(_SFR_ADDR(sfr))
#define _SFR_WORD(sfr) _MMIO_WORD(_SFR_ADDR(sfr))
#define _SFR_DWORD(sfr) _MMIO_DWORD(_SFR_ADDR(sfr))

/** \name Bit manipulation */

/**@{*/
/** \def _BV
    \ingroup avr_sfr

    \code #include <avr/io.h>\endcode

    Converts a bit number into a byte value.

    \note The bit shift is performed by the compiler which then inserts the
    result into the code. Thus, there is no run-time overhead when using
    _BV(). */

#define _BV(bit) (1 « (bit))

/**@}*/

#ifndef _VECTOR
#define _VECTOR(N) __vector_ ## N
#endif

#ifndef __ASSEMBLER__


/** \name IO register bit manipulation */
```

```

/**@{*/



/** \def bit_is_set
    \ingroup avr_sfr

    \code #include <avr/io.h>\endcode

    Test whether bit \c bit in IO register \c sfr is set.
    This will return a 0 if the bit is clear, and non-zero
    if the bit is set. */

#define bit_is_set(sfr, bit) (_SFR_BYTE(sfr) & _BV(bit))

/** \def bit_is_clear
    \ingroup avr_sfr

    \code #include <avr/io.h>\endcode

    Test whether bit \c bit in IO register \c sfr is clear.
    This will return non-zero if the bit is clear, and a 0
    if the bit is set. */

#define bit_is_clear(sfr, bit) (!(_SFR_BYTE(sfr) & _BV(bit)))

/** \def loop_until_bit_is_set
    \ingroup avr_sfr

    \code #include <avr/io.h>\endcode

    Wait until bit \c bit in IO register \c sfr is set. */

#define loop_until_bit_is_set(sfr, bit) do { } while (bit_is_clear(sfr, bit))

/** \def loop_until_bit_is_clear
    \ingroup avr_sfr

    \code #include <avr/io.h>\endcode

    Wait until bit \c bit in IO register \c sfr is clear. */

#define loop_until_bit_is_clear(sfr, bit) do { } while (bit_is_set(sfr, bit))

/**@}*/

#endif /* !__ASSEMBLER__ */

#endif  /* _SFR_DEFS_H_ */
```

23.31 signal.h

```
/* Copyright (c) 2002,2005,2006 Marek Michalkiewicz
   All rights reserved.

   Redistribution and use in source and binary forms, with or without
   modification, are permitted provided that the following conditions are met:

   * Redistributions of source code must retain the above copyright
     notice, this list of conditions and the following disclaimer.

   * Redistributions in binary form must reproduce the above copyright
     notice, this list of conditions and the following disclaimer in
     the documentation and/or other materials provided with the
     distribution.

   * Neither the name of the copyright holders nor the names of
     contributors may be used to endorse or promote products derived
     from this software without specific prior written permission.

  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
  AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
  IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
  ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
  LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
  CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
  SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
  INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
  CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
  ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
  POSSIBILITY OF SUCH DAMAGE. */

/* $Id$ */
```

```
#ifndef _AVR_SIGNAL_H_
#define _AVR_SIGNAL_H_

#warning "This header file is obsolete.  Use <avr/interrupt.h>."
#include <avr/interrupt.h>

#endif /* _AVR_SIGNAL_H_ */
```

23.32 signature.h File Reference

23.33 signature.h

Go to the documentation of this file.

```
/* Copyright (c) 2009, Atmel Corporation
   All rights reserved.

   Redistribution and use in source and binary forms, with or without
   modification, are permitted provided that the following conditions are met:

   * Redistributions of source code must retain the above copyright
     notice, this list of conditions and the following disclaimer.

   * Redistributions in binary form must reproduce the above copyright
     notice, this list of conditions and the following disclaimer in
     the documentation and/or other materials provided with the
     distribution.

   * Neither the name of the copyright holders nor the names of
     contributors may be used to endorse or promote products derived
     from this software without specific prior written permission.

  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
  AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
  IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
  ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
  LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
  CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
  SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
  INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
  CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
  ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
  POSSIBILITY OF SUCH DAMAGE. */

/* $Id$ */

/* avr/signature.h - Signature API */

#ifndef _AVR_SIGNATURE_H_
#define _AVR_SIGNATURE_H_ 1

/** \file */
/** \defgroup avr_signature <avr/signature.h>: Signature Support

    \par Introduction

    The <avr/signature.h> header file allows the user to automatically
    and easily include the device's signature data in a special section of
    the final linked ELF file.

    This value can then be used by programming software to compare the on-device
    signature with the signature recorded in the ELF file to look for a match
    before programming the device.

    \par API Usage Example

    Usage is very simple; just include the header file:

    \code
    #include <avr/signature.h>
    \endcode

    This will declare a constant unsigned char array and it is initialized with
    the three signature bytes, MSB first, that are defined in the device I/O
    header file. This array is then placed in the .signature section in the
    resulting linked ELF file.

    The three signature bytes that are used to initialize the array are
    these defined macros in the device I/O header file, from MSB to LSB:
    SIGNATURE_2, SIGNATURE_1, SIGNATURE_0.
```

```
   This header file should only be included once in an application.
*/

#ifndef __ASSEMBLER__

#include <avr/io.h>

#if defined(SIGNATURE_0) && defined(SIGNATURE_1) && defined(SIGNATURE_2)

const unsigned char __signature[3]
__attribute__((__used__, __section__(".signature"))) =
        { SIGNATURE_2, SIGNATURE_1, SIGNATURE_0 };

#endif  /* defined(SIGNATURE_0) && defined(SIGNATURE_1) && defined(SIGNATURE_2) */

#endif  /* __ASSEMBLER__ */

#endif  /* _AVR_SIGNATURE_H_ */
```

23.34 sleep.h File Reference

Functions

- void sleep_enable (void)
- void sleep_disable (void)
- void sleep_cpu (void)
- void sleep_mode (void)
- void sleep_bod_disable (void)

23.35 sleep.h

Go to the documentation of this file.

```
/* Copyright (c) 2002, 2004 Theodore A. Roth
   Copyright (c) 2004, 2007, 2008 Eric B. Weddington
   Copyright (c) 2005, 2006, 2007 Joerg Wunsch
   All rights reserved.

   Redistribution and use in source and binary forms, with or without
   modification, are permitted provided that the following conditions are met:

   * Redistributions of source code must retain the above copyright
     notice, this list of conditions and the following disclaimer.

   * Redistributions in binary form must reproduce the above copyright
     notice, this list of conditions and the following disclaimer in
     the documentation and/or other materials provided with the
     distribution.

   * Neither the name of the copyright holders nor the names of
     contributors may be used to endorse or promote products derived
     from this software without specific prior written permission.

   THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
   AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
   IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
   ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
   LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
   CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
   SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
   INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
   CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
   ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
   POSSIBILITY OF SUCH DAMAGE. */

/* $Id$ */

#ifndef _AVR_SLEEP_H_
#define _AVR_SLEEP_H_ 1

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


```

```
/** \file */

/** \defgroup avr_sleep <avr/sleep.h>: Power Management and Sleep Modes */
/**@{*/
/**
    \code #include <avr/sleep.h>\endcode

    Use of the \c SLEEP instruction can allow an application to reduce its
    power comsumption considerably. AVR devices can be put into different
    sleep modes. Refer to the datasheet for the details relating to the device
    you are using.

    There are several macros provided in this header file to actually
    put the device into sleep mode.  The simplest way is to optionally
    set the desired sleep mode using \c set_sleep_mode() (it usually
    defaults to idle mode where the CPU is put on sleep but all
    peripheral clocks are still running), and then call
    \c sleep_mode(). This macro automatically sets the sleep enable bit, goes
    to sleep, and clears the sleep enable bit.

    Example:
    \code
    #include <avr/sleep.h>

    ...
      set_sleep_mode(<mode>);
      sleep_mode();
    \endcode

    Note that unless your purpose is to completely lock the CPU (until a
    hardware reset), interrupts need to be enabled before going to sleep.

    As the \c sleep_mode() macro might cause race conditions in some
    situations, the individual steps of manipulating the sleep enable
    (SE) bit, and actually issuing the \c SLEEP instruction, are provided
    in the macros \c sleep_enable(), \c sleep_disable(), and
    \c sleep_cpu().  This also allows for test-and-sleep scenarios that
    take care of not missing the interrupt that will awake the device
    from sleep.

    Example:
    \code
    #include <avr/interrupt.h>
    #include <avr/sleep.h>

    ...
      set_sleep_mode(<mode>);
      cli();
      if (some_condition)
      {
        sleep_enable();
        sei();
        sleep_cpu();
        sleep_disable();
      }
      sei();
    \endcode

    This sequence ensures an atomic test of \c some_condition with
    interrupts being disabled.  If the condition is met, sleep mode
    will be prepared, and the \c SLEEP instruction will be scheduled
    immediately after an \c SEI instruction.  As the intruction right
    after the \c SEI is guaranteed to be executed before an interrupt
    could trigger, it is sure the device will really be put to sleep.

    Some devices have the ability to disable the Brown Out Detector (BOD) before
    going to sleep. This will also reduce power while sleeping. If the
    specific AVR device has this ability then an additional macro is defined:
    \c sleep_bod_disable(). This macro generates inlined assembly code
    that will correctly implement the timed sequence for disabling the BOD
    before sleeping. However, there is a limited number of cycles after the
    BOD has been disabled that the device can be put into sleep mode, otherwise
    the BOD will not truly be disabled. Recommended practice is to disable
    the BOD (\c sleep_bod_disable()), set the interrupts (\c sei()), and then
    put the device to sleep (\c sleep_cpu()), like so:

    \code
    #include <avr/interrupt.h>
    #include <avr/sleep.h>

    ...
      set_sleep_mode(<mode>);
      cli();
      if (some_condition)
      {
        sleep_enable();
        sleep_bod_disable();
```

```
        sei();
        sleep_cpu();
        sleep_disable();
      }
      sei();
    \endcode
*/


/* Define an internal sleep control register and an internal sleep enable bit mask. */
#if defined(SLEEP_CTRL)

    /* XMEGA devices */
    #define _SLEEP_CONTROL_REG  SLEEP_CTRL
    #define _SLEEP_ENABLE_MASK  SLEEP_SEN_bm
    #define _SLEEP_SMODE_GROUP_MASK  SLEEP_SMODE_gm

#elif defined(SLPCTRL)

    /* New xmega devices */
    #define _SLEEP_CONTROL_REG  SLPCTRL_CTRLA
    #define _SLEEP_ENABLE_MASK  SLPCTRL_SEN_bm
    #define _SLEEP_SMODE_GROUP_MASK  SLPCTRL_SMODE_gm

#elif defined(SMCR)

    #define _SLEEP_CONTROL_REG  SMCR
    #define _SLEEP_ENABLE_MASK  _BV(SE)

#elif defined(__AVR_AT94K__)

    #define _SLEEP_CONTROL_REG  MCUR
    #define _SLEEP_ENABLE_MASK  _BV(SE)

#elif !defined(__DOXYGEN__)

    #define _SLEEP_CONTROL_REG  MCUCR
    #define _SLEEP_ENABLE_MASK  _BV(SE)

#endif


/* Special casing these three devices - they are the
   only ones that need to write to more than one register. */
#if defined(__AVR_ATmega161__)

    #define set_sleep_mode(mode) \
    do { \
        MCUCR = ((MCUCR & ~_BV(SM1)) | ((mode) == SLEEP_MODE_PWR_DOWN || (mode) == SLEEP_MODE_PWR_SAVE ? _BV(SM1) : 0)); \
        EMCUCR = ((EMCUCR & ~_BV(SM0)) | ((mode) == SLEEP_MODE_PWR_SAVE ? _BV(SM0) : 0)); \
    } while(0)


#elif defined(__AVR_ATmega162__) \
|| defined(__AVR_ATmega8515__)

    #define set_sleep_mode(mode) \
    do { \
        MCUCR = ((MCUCR & ~_BV(SM1)) | ((mode) == SLEEP_MODE_IDLE ? 0 : _BV(SM1))); \
        MCUCSR = ((MCUCSR & ~_BV(SM2)) | ((mode) == SLEEP_MODE_STANDBY  || (mode) == SLEEP_MODE_EXT_STANDBY ? _BV(SM2) : 0)); \
        EMCUCR = ((EMCUCR & ~_BV(SM0)) | ((mode) == SLEEP_MODE_PWR_SAVE || (mode) == SLEEP_MODE_EXT_STANDBY ? _BV(SM0) : 0)); \
    } while(0)

/* For xmegas, check presence of SLEEP_SMODE<n>_bm and define set_sleep_mode accordingly. */
#elif defined(__AVR_XMEGA__)

#define set_sleep_mode(mode) \
  do { \
    _SLEEP_CONTROL_REG = ((_SLEEP_CONTROL_REG & ~(_SLEEP_SMODE_GROUP_MASK)) | (mode)); \
  } while(0)

/* For everything else, check for presence of SM<n> and define set_sleep_mode accordingly. */
#else
#if defined(SM2)

    #define set_sleep_mode(mode) \
    do { \
        _SLEEP_CONTROL_REG = ((_SLEEP_CONTROL_REG & ~(_BV(SM0) | _BV(SM1) | _BV(SM2))) | (mode)); \
    } while(0)

#elif defined(SM1)

    #define set_sleep_mode(mode) \
    do { \
```

```
        _SLEEP_CONTROL_REG = ((_SLEEP_CONTROL_REG & ~(_BV(SM0) | _BV(SM1))) | (mode)); \
    } while(0)

#elif defined(SM)

    #define set_sleep_mode(mode) \
    do { \
        _SLEEP_CONTROL_REG = ((_SLEEP_CONTROL_REG & ~_BV(SM)) | (mode)); \
    } while(0)

#else

    #error "No SLEEP mode defined for this device."

#endif /* if defined(SM2) */
#endif /* #if defined(__AVR_ATmega161__) */



/** \ingroup avr_sleep

    Put the device in sleep mode. How the device is brought out of sleep mode
    depends on the specific mode selected with the set_sleep_mode() function.
    See the data sheet for your device for more details. */


#if defined(__DOXYGEN__)

/** \ingroup avr_sleep

    Set the SE (sleep enable) bit.
*/
extern void sleep_enable (void);

#else

#define sleep_enable()             \
do {                               \
  _SLEEP_CONTROL_REG |= (uint8_t)_SLEEP_ENABLE_MASK;   \
} while(0)

#endif


#if defined(__DOXYGEN__)

/** \ingroup avr_sleep

    Clear the SE (sleep enable) bit.
*/
extern void sleep_disable (void);

#else

#define sleep_disable()            \
do {                               \
  _SLEEP_CONTROL_REG &= (uint8_t)(~_SLEEP_ENABLE_MASK);  \
} while(0)

#endif


/** \ingroup avr_sleep

    Put the device into sleep mode.  The SE bit must be set
    beforehand, and it is recommended to clear it afterwards.
*/
#if defined(__DOXYGEN__)

extern void sleep_cpu (void);

#else

#define sleep_cpu()                              \
do {                                             \
  __asm__ __volatile__ ( "sleep" "\n\t" :: );    \
} while(0)

#endif


#if defined(__DOXYGEN__)

/** \ingroup avr_sleep

    Put the device into sleep mode, taking care of setting
    the SE bit before, and clearing it afterwards. */
```

```
extern void sleep_mode (void);

#else

#define sleep_mode() \
do {                 \
    sleep_enable();  \
    sleep_cpu();     \
    sleep_disable(); \
} while (0)

#endif


#if defined(__DOXYGEN__)

/** \ingroup avr_sleep

    Disable BOD before going to sleep.
    Not available on all devices.
*/
extern void sleep_bod_disable (void);

#else

#if defined(BODS) && defined(BODSE)

#ifdef BODCR

#define BOD_CONTROL_REG BODCR

#else

#define BOD_CONTROL_REG MCUCR

#endif

#define sleep_bod_disable() \
do { \
  uint8_t tempreg; \
  __asm__ __volatile__("in %[tempreg], %[mcucr]" "\n\t" \
                       "ori %[tempreg], %[bods_bodse]" "\n\t" \
                       "out %[mcucr], %[tempreg]" "\n\t" \
                       "andi %[tempreg], %[not_bodse]" "\n\t" \
                       "out %[mcucr], %[tempreg]" \
                       : [tempreg] "=&d" (tempreg) \
                       : [mcucr] "I" _SFR_IO_ADDR(BOD_CONTROL_REG), \
                         [bods_bodse] "i" (_BV(BODS) | _BV(BODSE)), \
                         [not_bodse] "i" (~_BV(BODSE))); \
} while (0)

#endif

#endif


/**@}*/

#endif /* _AVR_SLEEP_H_ */
```

23.36 version.h

```
/* Copyright (c) 2005, Joerg Wunsch                               -*- c -*-
   All rights reserved.

   Redistribution and use in source and binary forms, with or without
   modification, are permitted provided that the following conditions are met:

   * Redistributions of source code must retain the above copyright
     notice, this list of conditions and the following disclaimer.

   * Redistributions in binary form must reproduce the above copyright
     notice, this list of conditions and the following disclaimer in
     the documentation and/or other materials provided with the
     distribution.

   * Neither the name of the copyright holders nor the names of
     contributors may be used to endorse or promote products derived
     from this software without specific prior written permission.

  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
  AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
  IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
  ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
```

```
  LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
  CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
  SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
  INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
  CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
  ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
  POSSIBILITY OF SUCH DAMAGE. */

/* $Id$ */

/** \defgroup avr_version <avr/version.h>: avr-libc version macros
    \code #include <avr/version.h> \endcode

    This header file defines macros that contain version numbers and
    strings describing the current version of avr-libc.

    The version number itself basically consists of three pieces that
    are separated by a dot: the major number, the minor number, and
    the revision number.  For development versions (which use an odd
    minor number), the string representation additionally gets the
    date code (YYYYMMDD) appended.

    This file will also be included by \c <avr/io.h>.  That way,
    portable tests can be implemented using \c <avr/io.h> that can be
    used in code that wants to remain backwards-compatible to library
    versions prior to the date when the library version API had been
    added, as referenced but undefined C preprocessor macros
    automatically evaluate to 0.
*/

#ifndef _AVR_VERSION_H_
#define _AVR_VERSION_H_

/** \ingroup avr_version
    String literal representation of the current library version. */
#define __AVR_LIBC_VERSION_STRING__ "2.2.0"

/** \ingroup avr_version
    Numerical representation of the current library version.

    In the numerical representation, the major number is multiplied by
    10000, the minor number by 100, and all three parts are then
    added.  It is intented to provide a monotonically increasing
    numerical value that can easily be used in numerical checks.
 */
#define __AVR_LIBC_VERSION__        20200UL

/** \ingroup avr_version
    String literal representation of the release date. */
#define __AVR_LIBC_DATE_STRING__    "20240608"

/** \ingroup avr_version
    Numerical representation of the release date. */
#define __AVR_LIBC_DATE_            20240608UL

/** \ingroup avr_version
    Library major version number. */
#define __AVR_LIBC_MAJOR__          2

/** \ingroup avr_version
    Library minor version number. */
#define __AVR_LIBC_MINOR__          2

/** \ingroup avr_version
    Library revision number. */
#define __AVR_LIBC_REVISION__       0

#endif /* _AVR_VERSION_H_ */
```

23.37 wdt.h File Reference

Macros

- #define wdt_reset() __asm__ __volatile__ ("wdr")
- #define wdt_enable(timeout)
- #define WDTO_15MS 0
- #define WDTO_30MS 1
- #define WDTO_60MS 2

- #define WDTO_120MS 3
- #define WDTO_250MS 4
- #define WDTO_500MS 5
- #define WDTO_1S 6
- #define WDTO_2S 7
- #define WDTO_4S 8
- #define WDTO_8S 9

Functions

- static void **wdt_enable** (const uint8_t value)
- static void **wdt_disable** (void)

23.38 wdt.h

Go to the documentation of this file.

```
/* Copyright (c) 2002, 2004 Marek Michalkiewicz
   Copyright (c) 2005, 2006, 2007 Eric B. Weddington
   All rights reserved.

   Redistribution and use in source and binary forms, with or without
   modification, are permitted provided that the following conditions are met:

   * Redistributions of source code must retain the above copyright
     notice, this list of conditions and the following disclaimer.

   * Redistributions in binary form must reproduce the above copyright
     notice, this list of conditions and the following disclaimer in
     the documentation and/or other materials provided with the
     distribution.

   * Neither the name of the copyright holders nor the names of
     contributors may be used to endorse or promote products derived
     from this software without specific prior written permission.

  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
  AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
  IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
  ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
  LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
  CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
  SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
  INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
  CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
  ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
  POSSIBILITY OF SUCH DAMAGE. */

/* $Id$ */

/*
   avr/wdt.h - macros for AVR watchdog timer
 */

#ifndef _AVR_WDT_H_
#define _AVR_WDT_H_

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

/** \file */
/** \defgroup avr_watchdog <avr/wdt.h>: Watchdog timer handling
    \code #include <avr/wdt.h> \endcode

    This header file declares the interface to some inline macros
    handling the watchdog timer present in many AVR devices.  In order
    to prevent the watchdog timer configuration from being
    accidentally altered by a crashing application, a special timed
    sequence is required in order to change it.  The macros within
    this header file handle the required sequence automatically
    before changing any value.  Interrupts will be disabled during
    the manipulation.

    \note Depending on the fuse configuration of the particular
    device, further restrictions might apply, in particular it might
```

```
   be disallowed to turn off the watchdog timer.

   Note that for newer devices (ATmega88 and newer, effectively any
   AVR that has the option to also generate interrupts), the watchdog
   timer remains active even after a system reset (except a power-on
   condition), using the fastest prescaler value (approximately 15
   ms).  It is therefore required to turn off the watchdog early
   during program startup, the datasheet recommends a sequence like
   the following:

   \code
   #include <stdint.h>
   #include <avr/wdt.h>

   uint8_t mcusr_mirror __attribute__ ((section (".noinit")));

   __attribute__((used, unused, naked, section(".init3")))
   static void get_mcusr (void);

   void get_mcusr (void)
   {
     mcusr_mirror = MCUSR;
     MCUSR = 0;
     wdt_disable();
   }
   \endcode

   Saving the value of MCUSR in \c mcusr_mirror is only needed if the
   application later wants to examine the reset source, but in particular,
   clearing the watchdog reset flag before disabling the
   watchdog is required, according to the datasheet.
*/

/**
  \ingroup avr_watchdog
  Reset the watchdog timer.  When the watchdog timer is enabled,
  a call to this instruction is required before the timer expires,
  otherwise a watchdog-initiated device reset will occur.
*/

#define wdt_reset() __asm__ __volatile__ ("wdr")

#ifndef __DOXYGEN__

#ifndef __ATTR_ALWAYS_INLINE__
#define __ATTR_ALWAYS_INLINE__ __inline__ __attribute__((__always_inline__))
#endif

#if defined(WDP3)
# define _WD_PS3_MASK       _BV(WDP3)
#else
# define _WD_PS3_MASK       0x00
#endif

#if defined(WDTCSR)
#  define _WD_CONTROL_REG     WDTCSR
#elif defined(WDTCR)
#  define _WD_CONTROL_REG     WDTCR
#else
#  define _WD_CONTROL_REG     WDT
#endif

#if defined(WDTOE)
#define _WD_CHANGE_BIT      WDTOE
#else
#define _WD_CHANGE_BIT      WDCE
#endif

#endif  /* !__DOXYGEN__ */

#ifdef __DOXYGEN__
/**
  \ingroup avr_watchdog
  Enable the watchdog timer, configuring it for expiry after
  \c timeout (which is a combination of the \c WDP0 through
  \c WDP2 bits to write into the \c WDTCR register; For those devices
  that have a \c WDTCSR register, it uses the combination of the \c WDP0
  through \c WDP3 bits).

  See also the symbolic constants \c WDTO_15MS et al.
*/
#define wdt_enable(timeout)
#endif  /* __DOXYGEN__ */


#if defined(__AVR_XMEGA__)

```

```
#if defined (WDT_CTRLA) && !defined(RAMPD)

#define wdt_enable(timeout) \
do { \
uint8_t __temp; \
__asm__ __volatile__ ( \
    "wdr"                                   "\n\t" \
    "out %[ccp_reg], %[ioreg_cen_mask]"     "\n\t" \
    "lds %[tmp], %[wdt_reg]"                "\n\t" \
    "sbr %[tmp], %[wdt_enable_timeout]"     "\n\t" \
    "sts %[wdt_reg], %[tmp]"                "\n\t" \
    "1:lds %[tmp], %[wdt_status_reg]"       "\n\t" \
    "sbrc %[tmp], %[wdt_syncbusy_bit]"      "\n\t" \
    "rjmp 1b"                                      \
    : [tmp]                  "=d"  (__temp) \
    : [ccp_reg]              "I"   (_SFR_IO_ADDR(CCP)), \
      [ioreg_cen_mask]       "r"   ((uint8_t)CCP_IOREG_gc), \
      [wdt_reg]              "n"   (_SFR_MEM_ADDR(WDT_CTRLA)), \
      [wdt_enable_timeout]   "M"   (timeout), \
      [wdt_status_reg]       "n"   (_SFR_MEM_ADDR(WDT_STATUS)), \
      [wdt_syncbusy_bit]     "I"   (WDT_SYNCBUSY_bm) \
); \
} while(0)

#define wdt_disable() \
do { \
uint8_t __temp; \
__asm__ __volatile__ (  \
    "wdr"                               "\n\t" \
    "out %[ccp_reg], %[ioreg_cen_mask]" "\n\t" \
    "lds %[tmp], %[wdt_reg]"            "\n\t" \
    "cbr %[tmp], %[timeout_mask]"       "\n\t" \
    "sts %[wdt_reg], %[tmp]"                   \
    : [tmp]             "=d" (__temp) \
    : [ccp_reg]         "I"  (_SFR_IO_ADDR(CCP)),        \
      [ioreg_cen_mask]  "r"  ((uint8_t)CCP_IOREG_gc),    \
      [wdt_reg]         "n"  (_SFR_MEM_ADDR(WDT_CTRLA)),\
      [timeout_mask]    "I"  (WDT_PERIOD_gm) \
); \
} while(0)

#else // defined (WDT_CTRLA) && !defined(RAMPD)

/*
   wdt_enable(timeout) for xmega devices
** write signature (CCP_IOREG_gc) that enables change of protected I/O
   registers to the CCP register
** At the same time,
   1) set WDT change enable (WDT_CEN_bm)
   2) enable WDT (WDT_ENABLE_bm)
   3) set timeout (timeout)
** Synchronization starts when ENABLE bit of WDT is set. So, wait till it
   finishes (SYNCBUSY of STATUS register is automatically cleared after the
   sync is finished).
*/
#define wdt_enable(timeout) \
do { \
uint8_t __temp; \
__asm__ __volatile__ (         \
    "in __tmp_reg__, %[rampd]"              "\n\t" \
    "out %[rampd], __zero_reg__"            "\n\t" \
    "out %[ccp_reg], %[ioreg_cen_mask]"     "\n\t" \
    "sts %[wdt_reg], %[wdt_enable_timeout]" "\n\t" \
    "1:lds %[tmp], %[wdt_status_reg]"       "\n\t" \
    "sbrc %[tmp], %[wdt_syncbusy_bit]"      "\n\t" \
    "rjmp 1b"                               "\n\t" \
    "out %[rampd], __tmp_reg__"                    \
    : [tmp]                "=r" (__temp) \
    : [rampd]              "I" (_SFR_IO_ADDR(RAMPD)),       \
      [ccp_reg]            "I" (_SFR_IO_ADDR(CCP)),         \
      [ioreg_cen_mask]     "r" ((uint8_t)CCP_IOREG_gc),     \
      [wdt_reg]            "n" (_SFR_MEM_ADDR(WDT_CTRL)),   \
      [wdt_enable_timeout] "r" ((uint8_t)(WDT_CEN_bm | WDT_ENABLE_bm | timeout)), \
      [wdt_status_reg]     "n" (_SFR_MEM_ADDR(WDT_STATUS)), \
      [wdt_syncbusy_bit]   "I" (WDT_SYNCBUSY_bm)            \
    : "r0" \
); \
} while(0)

#define wdt_disable() \
__asm__ __volatile__ (  \
    "in __tmp_reg__, %[rampd]"          "\n\t" \
    "out %[rampd], __zero_reg__"        "\n\t" \
    "out %[ccp_reg], %[ioreg_cen_mask]" "\n\t" \
    "sts %[wdt_reg], %[disable_mask]"   "\n\t" \
    "out %[rampd], __tmp_reg__"                \
    : /* no outputs */ \
```

```
    : [rampd]              "I" (_SFR_IO_ADDR(RAMPD)),    \
      [ccp_reg]            "I" (_SFR_IO_ADDR(CCP)),      \
      [ioreg_cen_mask]     "r" ((uint8_t)CCP_IOREG_gc),   \
      [wdt_reg]            "n" (_SFR_MEM_ADDR(WDT_CTRL)), \
      [disable_mask]       "r" ((uint8_t)((~WDT_ENABLE_bm) | WDT_CEN_bm)) \
    : "r0" \
)

#endif // defined (WDT_CTRLA) && !defined(RAMPD)

#elif defined(__AVR_TINY__)

#define wdt_enable(value) \
__asm__ __volatile__ ( \
    "in __tmp_reg__,__SREG__" "\n\t"  \
    "cli" "\n\t"  \
    "wdr" "\n\t"  \
    "out %[CCPADDRESS],%[SIGNATURE]" "\n\t"  \
    "out %[WDTREG],%[WDVALUE]" "\n\t"   \
    "out __SREG__,__tmp_reg__"          \
    : /* no outputs */  \
    : [CCPADDRESS] "I" (_SFR_IO_ADDR(CCP)),  \
      [SIGNATURE] "r" ((uint8_t)0xD8), \
      [WDTREG] "I" (_SFR_IO_ADDR(_WD_CONTROL_REG)), \
      [WDVALUE] "r" ((uint8_t)((value & 0x08 ? _WD_PS3_MASK : 0x00) \
      | _BV(WDE) | (value & 0x07) )) \
    : "r16" \
)

#define wdt_disable() \
do { \
uint8_t __temp_wd; \
__asm__ __volatile__ ( \
    "in __tmp_reg__,__SREG__" "\n\t"  \
    "cli" "\n\t"  \
    "wdr" "\n\t"  \
    "out %[CCPADDRESS],%[SIGNATURE]" "\n\t"  \
    "in  %[TEMP_WD],%[WDTREG]" "\n\t" \
    "cbr %[TEMP_WD],%[WDVALUE]" "\n\t" \
    "out %[WDTREG],%[TEMP_WD]" "\n\t" \
    "out __SREG__,__tmp_reg__"        \
    : [TEMP_WD] "=d" (__temp_wd) \
    : [CCPADDRESS] "I" (_SFR_IO_ADDR(CCP)), \
      [SIGNATURE] "r" ((uint8_t)0xD8), \
      [WDTREG] "I" (_SFR_IO_ADDR(_WD_CONTROL_REG)), \
      [WDVALUE] "n" (1 « WDE) \
    : "r16" \
); \
} while(0)

#elif defined(CCP)

static __ATTR_ALWAYS_INLINE__
void wdt_enable (const uint8_t value)
{
    if (!_SFR_IO_REG_P (CCP) && !_SFR_IO_REG_P (_WD_CONTROL_REG))
    {
        __asm__ __volatile__ (
            "in __tmp_reg__,__SREG__" "\n\t"
            "cli" "\n\t"
            "wdr" "\n\t"
            "sts %[CCPADDRESS],%[SIGNATURE]" "\n\t"
            "sts %[WDTREG],%[WDVALUE]" "\n\t"
            "out __SREG__,__tmp_reg__"
            : /* no outputs */
            : [CCPADDRESS] "n" (_SFR_MEM_ADDR(CCP)),
            [SIGNATURE] "r" ((uint8_t)0xD8),
            [WDTREG] "n" (_SFR_MEM_ADDR(_WD_CONTROL_REG)),
            [WDVALUE] "r" ((uint8_t)((value & 0x08 ? _WD_PS3_MASK : 0x00)
                | _BV(WDE) | (value & 0x07) ))
            : "r0"
            );
    }
    else if (!_SFR_IO_REG_P (CCP) && _SFR_IO_REG_P (_WD_CONTROL_REG))
    {
        __asm__ __volatile__ (
            "in __tmp_reg__,__SREG__" "\n\t"
            "cli" "\n\t"
            "wdr" "\n\t"
            "sts %[CCPADDRESS],%[SIGNATURE]" "\n\t"
            "out %[WDTREG],%[WDVALUE]" "\n\t"
            "out __SREG__,__tmp_reg__"
            : /* no outputs */
            : [CCPADDRESS] "n" (_SFR_MEM_ADDR(CCP)),
            [SIGNATURE] "r" ((uint8_t)0xD8),
            [WDTREG] "I" (_SFR_IO_ADDR(_WD_CONTROL_REG)),
            [WDVALUE] "r" ((uint8_t)((value & 0x08 ? _WD_PS3_MASK : 0x00)
```

```
                | _BV(WDE) | (value & 0x07) ))
            : "r0"
            );
    }
    else if (_SFR_IO_REG_P (CCP) && !_SFR_IO_REG_P (_WD_CONTROL_REG))
    {
        __asm__ __volatile__ (
            "in __tmp_reg__,__SREG__" "\n\t"
            "cli" "\n\t"
            "wdr" "\n\t"
            "out %[CCPADDRESS],%[SIGNATURE]" "\n\t"
            "sts %[WDTREG],%[WDVALUE]" "\n\t"
            "out __SREG__,__tmp_reg__"
            : /* no outputs */
            : [CCPADDRESS] "I" (_SFR_IO_ADDR(CCP)),
            [SIGNATURE] "r" ((uint8_t)0xD8),
            [WDTREG] "n" (_SFR_MEM_ADDR(_WD_CONTROL_REG)),
            [WDVALUE] "r" ((uint8_t)((value & 0x08 ? _WD_PS3_MASK : 0x00)
                | _BV(WDE) | (value & 0x07) ))
            : "r0"
            );
    }
    else
    {
        __asm__ __volatile__ (
            "in __tmp_reg__,__SREG__" "\n\t"
            "cli" "\n\t"
            "wdr" "\n\t"
            "out %[CCPADDRESS],%[SIGNATURE]" "\n\t"
            "out %[WDTREG],%[WDVALUE]" "\n\t"
            "out __SREG__,__tmp_reg__"
            : /* no outputs */
            : [CCPADDRESS] "I" (_SFR_IO_ADDR(CCP)),
            [SIGNATURE] "r" ((uint8_t)0xD8),
            [WDTREG] "I" (_SFR_IO_ADDR(_WD_CONTROL_REG)),
            [WDVALUE] "r" ((uint8_t)((value & 0x08 ? _WD_PS3_MASK : 0x00)
                | _BV(WDE) | (value & 0x07) ))
            : "r0"
            );
    }
}

static __ATTR_ALWAYS_INLINE__
void wdt_disable (void)
{
    if (!_SFR_IO_REG_P (CCP) && !_SFR_IO_REG_P(_WD_CONTROL_REG))
    {
        uint8_t __temp_wd;
        __asm__ __volatile__ (
                "in __tmp_reg__,__SREG__" "\n\t"
                "cli" "\n\t"
                "wdr" "\n\t"
                "sts %[CCPADDRESS],%[SIGNATURE]" "\n\t"
                "lds %[TEMP_WD],%[WDTREG]" "\n\t"
                "cbr %[TEMP_WD],%[WDVALUE]" "\n\t"
                "sts %[WDTREG],%[TEMP_WD]" "\n\t"
                "out __SREG__,__tmp_reg__"
                : [TEMP_WD] "=d" (__temp_wd)
                : [CCPADDRESS] "n" (_SFR_MEM_ADDR(CCP)),
                [SIGNATURE] "r" ((uint8_t)0xD8),
                [WDTREG] "n" (_SFR_MEM_ADDR(_WD_CONTROL_REG)),
                [WDVALUE] "n" (1 « WDE)
                : "r0"
                );
    }
    else if (!_SFR_IO_REG_P (CCP) && _SFR_IO_REG_P(_WD_CONTROL_REG))
    {
        uint8_t __temp_wd;
        __asm__ __volatile__ (
                "in __tmp_reg__,__SREG__" "\n\t"
                "cli" "\n\t"
                "wdr" "\n\t"
                "sts %[CCPADDRESS],%[SIGNATURE]" "\n\t"
                "in %[TEMP_WD],%[WDTREG]" "\n\t"
                "cbr %[TEMP_WD],%[WDVALUE]" "\n\t"
                "out %[WDTREG],%[TEMP_WD]" "\n\t"
                "out __SREG__,__tmp_reg__"
                : [TEMP_WD] "=d" (__temp_wd)
                : [CCPADDRESS] "n" (_SFR_MEM_ADDR(CCP)),
                [SIGNATURE] "r" ((uint8_t)0xD8),
                [WDTREG] "I" (_SFR_IO_ADDR(_WD_CONTROL_REG)),
                [WDVALUE] "n" (1 « WDE)
                : "r0"
                );
    }
    else if (_SFR_IO_REG_P (CCP) && !_SFR_IO_REG_P(_WD_CONTROL_REG))
    {
```

```
        uint8_t __temp_wd;
        __asm__ __volatile__ (
                "in __tmp_reg__,__SREG__" "\n\t"
                "cli" "\n\t"
                "wdr" "\n\t"
                "out %[CCPADDRESS],%[SIGNATURE]" "\n\t"
                "lds %[TEMP_WD],%[WDTREG]" "\n\t"
                "cbr %[TEMP_WD],%[WDVALUE]" "\n\t"
                "sts %[WDTREG],%[TEMP_WD]" "\n\t"
                "out __SREG__,__tmp_reg__"
                : [TEMP_WD] "=d" (__temp_wd)
                : [CCPADDRESS] "I" (_SFR_IO_ADDR(CCP)),
                [SIGNATURE] "r" ((uint8_t)0xD8),
                [WDTREG] "n" (_SFR_MEM_ADDR(_WD_CONTROL_REG)),
                [WDVALUE] "n" (1 « WDE)
                : "r0"
                );
    }
    else
    {
        uint8_t __temp_wd;
        __asm__ __volatile__ (
                "in __tmp_reg__,__SREG__" "\n\t"
                "cli" "\n\t"
                "wdr" "\n\t"
                "out %[CCPADDRESS],%[SIGNATURE]" "\n\t"
                "in %[TEMP_WD],%[WDTREG]" "\n\t"
                "cbr %[TEMP_WD],%[WDVALUE]" "\n\t"
                "out %[WDTREG],%[TEMP_WD]" "\n\t"
                "out __SREG__,__tmp_reg__"
                : [TEMP_WD] "=d" (__temp_wd)
                : [CCPADDRESS] "I" (_SFR_IO_ADDR(CCP)),
                [SIGNATURE] "r" ((uint8_t)0xD8),
                [WDTREG] "I" (_SFR_IO_ADDR(_WD_CONTROL_REG)),
                [WDVALUE] "n" (1 « WDE)
                : "r0"
                );
    }
}

#else

static __ATTR_ALWAYS_INLINE__
void wdt_enable (const uint8_t value)
{
    if (_SFR_IO_REG_P (_WD_CONTROL_REG))
    {
        __asm__ __volatile__ (
                "in __tmp_reg__,__SREG__" "\n\t"
                "cli" "\n\t"
                "wdr" "\n\t"
                "out %0, %1" "\n\t"
                "out __SREG__,__tmp_reg__" "\n\t"
                "out %0, %2"
                : /* no outputs */
                : "I" (_SFR_IO_ADDR(_WD_CONTROL_REG)),
                "r" ((uint8_t)(_BV(_WD_CHANGE_BIT) | _BV(WDE))),
                "r" ((uint8_t) ((value & 0x08 ? _WD_PS3_MASK : 0x00) |
                        _BV(WDE) | (value & 0x07)) )
                : "r0"
        );
    }
    else
    {
        __asm__ __volatile__ (
                "in __tmp_reg__,__SREG__" "\n\t"
                "cli" "\n\t"
                "wdr" "\n\t"
                "sts %0, %1" "\n\t"
                "out __SREG__,__tmp_reg__" "\n\t"
                "sts %0, %2"
                : /* no outputs */
                : "n" (_SFR_MEM_ADDR(_WD_CONTROL_REG)),
                "r" ((uint8_t)(_BV(_WD_CHANGE_BIT) | _BV(WDE))),
                "r" ((uint8_t) ((value & 0x08 ? _WD_PS3_MASK : 0x00) |
                        _BV(WDE) | (value & 0x07)) )
                : "r0"
        );
    }
}

static __ATTR_ALWAYS_INLINE__
void wdt_disable (void)
{
    if (_SFR_IO_REG_P (_WD_CONTROL_REG))
    {
        uint8_t __temp_reg;
```

```
        __asm__ __volatile__ (
                "in __tmp_reg__,__SREG__"    "\n\t"
                "cli"                        "\n\t"
                "wdr"                        "\n\t"
                "in  %[TEMPREG],%[WDTREG]"   "\n\t"
                "ori %[TEMPREG],%[WDCE_WDE]" "\n\t"
                "out %[WDTREG],%[TEMPREG]"   "\n\t"
                "out %[WDTREG],__zero_reg__" "\n\t"
                "out __SREG__,__tmp_reg__"
                : [TEMPREG] "=d" (__temp_reg)
                : [WDTREG]  "I"  (_SFR_IO_ADDR(_WD_CONTROL_REG)),
                [WDCE_WDE]  "n"  ((uint8_t)(_BV(_WD_CHANGE_BIT) | _BV(WDE)))
                : "r0"
        );
    }
    else
    {
        uint8_t __temp_reg;
        __asm__ __volatile__ (
                "in __tmp_reg__,__SREG__"    "\n\t"
                "cli"                        "\n\t"
                "wdr"                        "\n\t"
                "lds %[TEMPREG],%[WDTREG]"   "\n\t"
                "ori %[TEMPREG],%[WDCE_WDE]" "\n\t"
                "sts %[WDTREG],%[TEMPREG]"   "\n\t"
                "sts %[WDTREG],__zero_reg__" "\n\t"
                "out __SREG__,__tmp_reg__"
                : [TEMPREG] "=d" (__temp_reg)
                : [WDTREG]  "n"  (_SFR_MEM_ADDR(_WD_CONTROL_REG)),
                [WDCE_WDE]  "n"  ((uint8_t)(_BV(_WD_CHANGE_BIT) | _BV(WDE)))
                : "r0"
        );
    }
}

#endif


/**
   \ingroup avr_watchdog
   Symbolic constants for the watchdog timeout.  Since the watchdog
   timer is based on a free-running RC oscillator, the times are
   approximate only and apply to a supply voltage of 5 V.  At lower
   supply voltages, the times will increase.  For older devices, the
   times will be as large as three times when operating at Vcc = 3 V,
   while the newer devices (e. g. ATmega128, ATmega8) only experience
   a negligible change.

   Possible timeout values are: 15 ms, 30 ms, 60 ms, 120 ms, 250 ms,
   500 ms, 1 s, 2 s.  (Some devices also allow for 4 s and 8 s.)
   Symbolic constants are formed by the prefix
   \c WDTO_, followed by the time.

   Example that would select a watchdog timer expiry of approximately
   500 ms:
   \code
   wdt_enable(WDTO_500MS);
   \endcode
*/
#define WDTO_15MS   0

/** \ingroup avr_watchdog
    See \c WDTO_15MS */
#define WDTO_30MS   1

/** \ingroup avr_watchdog
    See \c WDTO_15MS */
#define WDTO_60MS   2

/** \ingroup avr_watchdog
    See \c WDTO_15MS */
#define WDTO_120MS  3

/** \ingroup avr_watchdog
    See \c WDTO_15MS */
#define WDTO_250MS  4

/** \ingroup avr_watchdog
    See \c WDTO_15MS */
#define WDTO_500MS  5

/** \ingroup avr_watchdog
    See \c WDTO_15MS */
#define WDTO_1S     6

/** \ingroup avr_watchdog
    See \c WDTO_15MS */
```

```
#define WDTO_2S     7

#if defined(__DOXYGEN__) || defined(WDP3)

/** \ingroup avr_watchdog
    See \c WDTO_15MS
    Note: This is only available on the
    ATtiny2313,
    ATtiny24, ATtiny44, ATtiny84, ATtiny84A,
    ATtiny25, ATtiny45, ATtiny85,
    ATtiny261, ATtiny461, ATtiny861,
    ATmega48*, ATmega88*, ATmega168*, ATmega328*,
    ATmega164P, ATmega324P, ATmega324PB, ATmega644P, ATmega644,
    ATmega640, ATmega1280, ATmega1281, ATmega2560, ATmega2561,
    ATmega8HVA, ATmega16HVA, ATmega32HVB,
    ATmega406, ATmega1284P,
    AT90PWM1, AT90PWM2, AT90PWM2B, AT90PWM3, AT90PWM3B, AT90PWM216, AT90PWM316,
    AT90PWM81, AT90PWM161,
    AT90USB82, AT90USB162,
    AT90USB646, AT90USB647, AT90USB1286, AT90USB1287,
    ATtiny48, ATtiny88.

    Note: This value does <em>not</em> match the bit pattern of the
    respective control register.  It is solely meant to be used together
    with wdt_enable().
    */
#define WDTO_4S     8

/** \ingroup avr_watchdog
    See \c WDTO_15MS
    Note: This is only available on the
    ATtiny2313,
    ATtiny24, ATtiny44, ATtiny84, ATtiny84A,
    ATtiny25, ATtiny45, ATtiny85,
    ATtiny261, ATtiny461, ATtiny861,
    ATmega48*, ATmega88*, ATmega168*, ATmega328*,
    ATmega164P, ATmega324P, ATmega324PB, ATmega644P, ATmega644,
    ATmega640, ATmega1280, ATmega1281, ATmega2560, ATmega2561,
    ATmega8HVA, ATmega16HVA, ATmega32HVB,
    ATmega406, ATmega1284P,
    ATmega2564RFR2, ATmega256RFR2, ATmega1284RFR2, ATmega128RFR2, ATmega644RFR2, ATmega64RFR2
    AT90PWM1, AT90PWM2, AT90PWM2B, AT90PWM3, AT90PWM3B, AT90PWM216, AT90PWM316,
    AT90PWM81, AT90PWM161,
    AT90USB82, AT90USB162,
    AT90USB646, AT90USB647, AT90USB1286, AT90USB1287,
    ATtiny48, ATtiny88,
    ATxmega16a4u, ATxmega32a4u,
    ATxmega16c4, ATxmega32c4,
    ATxmega128c3, ATxmega192c3, ATxmega256c3.

    Note: This value does <em>not</em> match the bit pattern of the
    respective control register.  It is solely meant to be used together
    with wdt_enable().
    */
#define WDTO_8S     9

#endif  /* defined(__DOXYGEN__) || defined(WDP3) */


#endif /* _AVR_WDT_H_ */
```

23.39 xmega.h

```
/* Copyright (c) 2012 Joerg Wunsch
   All rights reserved.

   Redistribution and use in source and binary forms, with or without
   modification, are permitted provided that the following conditions are met:

   * Redistributions of source code must retain the above copyright
     notice, this list of conditions and the following disclaimer.

   * Redistributions in binary form must reproduce the above copyright
     notice, this list of conditions and the following disclaimer in
     the documentation and/or other materials provided with the
     distribution.

   * Neither the name of the copyright holders nor the names of
     contributors may be used to endorse or promote products derived
     from this software without specific prior written permission.

  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
  AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
  IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
```

```
  ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
  LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
  CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
  SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
  INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
  CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
  ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
  POSSIBILITY OF SUCH DAMAGE. */

/* $Id$ */

/*
 * This file is included by <avr/io.h> whenever compiling for an Xmega
 * device.  It abstracts certain features common to the Xmega device
 * families.
 */

#ifndef _AVR_XMEGA_H
#define _AVR_XMEGA_H

#ifdef __DOXYGEN__
/**
 \def _PROTECTED_WRITE
 \ingroup avr_io

 Write value \c value to IO register \c reg that is protected through
 the Xmega configuration change protection (CCP) mechanism.  This
 implements the timed sequence that is required for CCP.

 Example to modify the CPU clock:
 \code
 #include <avr/io.h>

 _PROTECTED_WRITE(CLK_PSCTRL, CLK_PSADIV0_bm);
 _PROTECTED_WRITE(CLK_CTRL, CLK_SCLKSEL0_bm);
 \endcode
 */
#define _PROTECTED_WRITE(reg, value)

/**
 \def _PROTECTED_WRITE_SPM
 \ingroup avr_io

 Write value \c value to register \c reg that is protected through
 the Xmega configuration change protection (CCP) key for self
 programming (SPM).  This implements the timed sequence that is
 required for CCP.

 Example to modify the CPU clock:
 \code
 #include <avr/io.h>

 _PROTECTED_WRITE_SPM(NVMCTRL_CTRLA, NVMCTRL_CMD_PAGEERASEWRITE_gc);
 \endcode
 */
#define _PROTECTED_WRITE_SPM(reg, value)

#else  /* !__DOXYGEN__ */

#define _PROTECTED_WRITE(reg, value)                          \
  __asm__ __volatile__("out %[ccp], %[ccp_ioreg]" "\n\t"    \
               "sts %[ioreg], %[val]"          \
               :                    \
               : [ccp] "I" (_SFR_IO_ADDR(CCP)),      \
             [ccp_ioreg] "d" ((uint8_t)CCP_IOREG_gc),    \
             [ioreg] "n" (_SFR_MEM_ADDR(reg)),   \
             [val] "r" ((uint8_t)value))

#define _PROTECTED_WRITE_SPM(reg, value) \
  __asm__ __volatile__("out %[ccp], %[ccp_spm_mask]" "\n\t" \
                       "sts %[ioreg], %[val]"               \
                       :                                    \
                       : [ccp]          "I" (_SFR_IO_ADDR(CCP)), \
                         [ccp_spm_mask] "d" ((uint8_t)CCP_SPM_gc), \
                         [ioreg]        "n" (_SFR_MEM_ADDR(reg)), \
                         [val]          "r" ((uint8_t)value))
#endif /* DOXYGEN */

#endif /* _AVR_XMEGA_H */
```

23.40 deprecated.h

```
/* Copyright (c) 2005,2006 Joerg Wunsch
   All rights reserved.
```

```

   Redistribution and use in source and binary forms, with or without
   modification, are permitted provided that the following conditions are met:

   * Redistributions of source code must retain the above copyright
     notice, this list of conditions and the following disclaimer.

   * Redistributions in binary form must reproduce the above copyright
     notice, this list of conditions and the following disclaimer in
     the documentation and/or other materials provided with the
     distribution.

   * Neither the name of the copyright holders nor the names of
     contributors may be used to endorse or promote products derived
     from this software without specific prior written permission.

  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
  AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
  IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
  ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
  LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
  CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
  SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
  INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
  CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
  ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
  POSSIBILITY OF SUCH DAMAGE. */

/* $Id$ */

#ifndef _COMPAT_DEPRECATED_H_
#define _COMPAT_DEPRECATED_H_

/** \defgroup deprecated_items <compat/deprecated.h>: Deprecated items

    This header file contains several items that used to be available
    in previous versions of this library, but have eventually been
    deprecated over time.

    \code #include <compat/deprecated.h> \endcode

    These items are supplied within that header file for backward
    compatibility reasons only, so old source code that has been
    written for previous library versions could easily be maintained
    until its end-of-life.  Use of any of these items in new code is
    strongly discouraged.
  */

/** \name Allowing specific system-wide interrupts

    In addition to globally enabling interrupts, each device's particular
    interrupt needs to be enabled separately if interrupts for this device are
    desired.  While some devices maintain their interrupt enable bit inside
    the device's register set, external and timer interrupts have system-wide
    configuration registers.

    Example:

    \code
    // Enable timer 1 overflow interrupts.
    timer_enable_int(_BV(TOIE1));

    // Do some work...

    // Disable all timer interrupts.
    timer_enable_int(0);
    \endcode

    \note Be careful when you use these functions. If you already have a
    different interrupt enabled, you could inadvertantly disable it by
    enabling another interrupt. */

/**@{*/

/** \ingroup deprecated_items
    \def enable_external_int(mask)
    \deprecated

    This macro gives access to the \c GIMSK register (or \c EIMSK register
    if using an AVR Mega device or \c GICR register for others). Although this
    macro is essentially the same as assigning to the register, it does
    adapt slightly to the type of device being used. This macro is
    unavailable if none of the registers listed above are defined. */

/* Define common register definition if available. */
#if defined(EIMSK)
#  define __EICR  EIMSK
```

```
#elif defined(GIMSK)
#  define __EICR  GIMSK
#elif defined(GICR)
#  define __EICR  GICR
#endif

/* If common register defined, define macro. */
#if defined(__EICR) || defined(__DOXYGEN__)
#define enable_external_int(mask)               (__EICR = mask)
#endif

/** \ingroup deprecated_items
    \deprecated

    This function modifies the \c timsk register.
    The value you pass via \c ints is device specific. */

static __inline__ void timer_enable_int (unsigned char ints)
{
#ifdef TIMSK
    TIMSK = ints;
#endif
}

/** \def INTERRUPT(signame)
    \ingroup deprecated_items
    \deprecated

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

    As this macro has been used by too many unsuspecting people in the
    past, it has been deprecated, and will be removed in a future
    version of the library.  Users who want to legitimately re-enable
    interrupts in their interrupt handlers as quickly as possible are
    encouraged to explicitly declare their handlers as described
    \ref attr_interrupt "above".
*/

#if (__GNUC__ == 4 && __GNUC_MINOR__ >= 1) || (__GNUC__ > 4)
#  define __INTR_ATTRS __used__, __externally_visible__
#else /* GCC < 4.1 */
#  define __INTR_ATTRS __used__
#endif

#ifdef __cplusplus
#define INTERRUPT(signame)                          \
extern "C" void signame(void);                      \
void signame (void) __attribute__ ((__interrupt__,__INTR_ATTRS));   \
void signame (void)
#else
#define INTERRUPT(signame)                          \
void signame (void) __attribute__ ((__interrupt__,__INTR_ATTRS));   \
void signame (void)
#endif

/**@}*/

/**
   \name Obsolete IO macros

   Back in a time when AVR-GCC and AVR-LibC could not handle IO port
   access in the direct assignment form as they are handled now, all
   IO port access had to be done through specific macros that
   eventually resulted in inline assembly instructions performing the
   desired action.

   These macros became obsolete, as reading and writing IO ports can
   be done by simply using the IO port name in an expression, and all
   bit manipulation (including those on IO ports) can be done using
   generic C bit manipulation operators.

   The macros in this group simulate the historical behaviour.  While
   they are supposed to be applied to IO ports, the emulation actually
   uses standard C methods, so they could be applied to arbitrary
   memory locations as well.
*/

/**@{*/

/**
   \ingroup deprecated_items
   \def inp(port)
   \deprecated

   Read a value from an IO port \c port.
*/
```

```
#define inp(port) (port)

/**
   \ingroup deprecated_items
   \def outp(val, port)
   \deprecated

   Write \c val to IO port \c port.
*/
#define outp(val, port) (port) = (val)

/**
   \ingroup deprecated_items
   \def inb(port)
   \deprecated

   Read a value from an IO port \c port.
*/
#define inb(port) (port)

/**
   \ingroup deprecated_items
   \def outb(port, val)
   \deprecated

   Write \c val to IO port \c port.
*/
#define outb(port, val) (port) = (val)

/**
   \ingroup deprecated_items
   \def sbi(port, bit)
   \deprecated

   Set \c bit in IO port \c port.
*/
#define sbi(port, bit) (port) |= (1 « (bit))

/**
   \ingroup deprecated_items
   \def cbi(port, bit)
   \deprecated

   Clear \c bit in IO port \c port.
*/
#define cbi(port, bit) (port) &= ~(1 « (bit))

/**@}*/

#endif /* _COMPAT_DEPRECATED_H_ */
```

23.41 ina90.h

```
/* Copyright (c) 2002,2004 Marek Michalkiewicz
   All rights reserved.

   Redistribution and use in source and binary forms, with or without
   modification, are permitted provided that the following conditions are met:

   * Redistributions of source code must retain the above copyright
     notice, this list of conditions and the following disclaimer.

   * Redistributions in binary form must reproduce the above copyright
     notice, this list of conditions and the following disclaimer in
     the documentation and/or other materials provided with the
     distribution.

   * Neither the name of the copyright holders nor the names of
     contributors may be used to endorse or promote products derived
     from this software without specific prior written permission.

  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
  AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
  IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
  ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
  LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
  CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
  SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
  INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
  CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
  ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
  POSSIBILITY OF SUCH DAMAGE. */

/* $Id$ */
```

```
/* copied from: Id: avr/ina90.h,v 1.8 2004/11/09 19:16:09 arcanum Exp */

/*
   ina90.h

   Contributors:
     Created by Marek Michalkiewicz <marekm@linux.org.pl>
 */

/**
   \defgroup compat_ina90 <compat/ina90.h>: Compatibility with IAR EWB 3.x

   \code #include <compat/ina90.h> \endcode

   This is an attempt to provide some compatibility with
   header files that come with IAR C, to make porting applications
   between different compilers easier.  No 100% compatibility though.

   \note For actual documentation, please see the IAR manual.
 */

#ifndef _INA90_H_
#define _INA90_H_ 1

#define _CLI() do { __asm__ __volatile__ ("cli"); } while (0)
#define _SEI() do { __asm__ __volatile__ ("sei"); } while (0)
#define _NOP() do { __asm__ __volatile__ ("nop"); } while (0)
#define _WDR() do { __asm__ __volatile__ ("wdr"); } while (0)
#define _SLEEP() do { __asm__ __volatile__ ("sleep"); } while (0)
#define _OPC(op) do { __asm__ __volatile__ (".word %0" : : "n" (op)); } while (0)

/* _LPM, _ELPM */
#include <avr/pgmspace.h>
#define _LPM(x) do { __LPM(x); } while (0)
#define _ELPM(x) do { __ELPM(x); } while (0)

/* _EEGET, _EEPUT */
#include <avr/eeprom.h>

#define input(port) (port)
#define output(port, val) do { (port) = (val); } while (0)

#define __inp_blk__(port, addr, cnt, op) do {   \
    unsigned char __i = (cnt);       \
    unsigned char *__addr = (addr);      \
    while (__i) {                \
        *(__addr op) = input(port); \
        __i--;                  \
    }                   \
 } while (0)

#define input_block_inc(port, addr, cnt) __inp_blk__(port, addr, cnt, ++)
#define input_block_dec(port, addr, cnt) __inp_blk__(port, addr, cnt, --)

#define __out_blk__(port, addr, cnt, op) do {   \
    unsigned char __i = (cnt);       \
    const unsigned char *__addr = (addr);    \
    while (__i) {                \
        output(port, *(__addr op)); \
        __i--;                  \
    }                   \
 } while (0)

#define output_block_inc(port, addr, cnt) __out_blk__(port, addr, cnt, ++)
#define output_block_dec(port, addr, cnt) __out_blk__(port, addr, cnt, --)

#endif

```

23.42 ctype.h File Reference

Functions

Character classification routines

These functions perform character classification. They return true or false status depending whether the character passed to the function falls into the function's classification (i.e. isdigit() returns true if its argument is any value '0' though '9', inclusive). If the input is not an unsigned char value, all of this function return false.

- int isalnum (int __c)

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

Character convertion routines

This realization permits all possible values of integer argument. The toascii() function clears all highest bits. The tolower() and toupper() functions return an input argument as is, if it is not an unsigned char value.

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

23.43 ctype.h

Go to the documentation of this file.

```
/* Copyright (c) 2002,2007 Michael Stumpf
   All rights reserved.

   Redistribution and use in source and binary forms, with or without
   modification, are permitted provided that the following conditions are met:

   * Redistributions of source code must retain the above copyright
     notice, this list of conditions and the following disclaimer.

   * Redistributions in binary form must reproduce the above copyright
     notice, this list of conditions and the following disclaimer in
     the documentation and/or other materials provided with the
     distribution.

   * Neither the name of the copyright holders nor the names of
     contributors may be used to endorse or promote products derived
     from this software without specific prior written permission.

  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
  AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
  IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
  ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
  LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
  CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
  SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
  INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
  CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
  ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
  POSSIBILITY OF SUCH DAMAGE. */

/* $Id$ */

/*
   ctype.h - character conversion macros and ctype macros

  Author : Michael Stumpf
           Michael.Stumpf@t-online.de
*/

#ifndef __CTYPE_H_
#define __CTYPE_H_ 1

#ifdef __cplusplus
extern "C" {
#endif

/** \file */
/** \defgroup ctype <ctype.h>: Character Operations
    These functions perform various operations on characters.

```

```
    \code #include <ctype.h>\endcode

*/

/** \name Character classification routines

    These functions perform character classification. They return true or
    false status depending whether the character passed to the function falls
    into the function's classification (i.e. isdigit() returns true if its
    argument is any value '0' though '9', inclusive). If the input is not
    an unsigned char value, all of this function return false.  */

/**@{*/

/** \ingroup ctype

    Checks for an alphanumeric character. It is equivalent to <tt>(isalpha(c)
    || isdigit(c))</tt>. */

extern int isalnum(int __c);

/** \ingroup ctype

    Checks for an alphabetic character. It is equivalent to <tt>(isupper(c) ||
    islower(c))</tt>. */

extern int isalpha(int __c);

/** \ingroup ctype

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

extern int isascii(int __c);

/** \ingroup ctype

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

extern int isblank(int __c);

/** \ingroup ctype

    Checks for a control character. */

extern int iscntrl(int __c);

/** \ingroup ctype

    Checks for a digit (0 through 9). */

extern int isdigit(int __c);

/** \ingroup ctype

    Checks for any printable character except space. */

extern int isgraph(int __c);

/** \ingroup ctype

    Checks for a lower-case character. */

extern int islower(int __c);

/** \ingroup ctype

    Checks for any printable character including space. */

extern int isprint(int __c);

/** \ingroup ctype

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

extern int ispunct(int __c);

/** \ingroup ctype

    Checks for white-space characters.  For the AVR-LibC library, these are:
    space, form-feed ('\\f'), newline ('\\n'), carriage return ('\\r'),
    horizontal tab ('\\t'), and vertical tab ('\\v'). */

extern int isspace(int __c);

/** \ingroup ctype
```

```

    Checks for an uppercase letter. */

extern int isupper(int __c);

/** \ingroup ctype

    Checks for a hexadecimal digits, i.e. one of 0 1 2 3 4 5 6 7 8 9 a b c d e
    f A B C D E F. */

extern int isxdigit(int __c);

/**@}*/

/** \name Character convertion routines

    This realization permits all possible values of integer argument.
    The toascii() function clears all highest bits. The tolower() and
    toupper() functions return an input argument as is, if it is not an
    unsigned char value.    */

/**@{*/

/** \ingroup ctype

    Converts \c c to a 7-bit unsigned char value that fits into the ASCII
    character set, by clearing the high-order bits.

    \warning Many people will be unhappy if you use this function. This
    function will convert accented letters into random characters. */

extern int toascii(int __c);

/** \ingroup ctype

    Converts the letter \c c to lower case, if possible. */

extern int tolower(int __c);

/** \ingroup ctype

    Converts the letter \c c to upper case, if possible. */

extern int toupper(int __c);

/**@}*/

#ifdef __cplusplus
}
#endif

#endif
```

23.44 errno.h File Reference

Macros

- #define EDOM 33
- #define ERANGE 34

Variables

- int errno

23.45 errno.h

Go to the documentation of this file.

```
/* Copyright (c) 2002,2007 Marek Michalkiewicz
   All rights reserved.

   Redistribution and use in source and binary forms, with or without
   modification, are permitted provided that the following conditions are met:
```

```
   * Redistributions of source code must retain the above copyright
     notice, this list of conditions and the following disclaimer.

   * Redistributions in binary form must reproduce the above copyright
     notice, this list of conditions and the following disclaimer in
     the documentation and/or other materials provided with the
     distribution.

   * Neither the name of the copyright holders nor the names of
     contributors may be used to endorse or promote products derived
     from this software without specific prior written permission.

  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
  AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
  IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
  ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
  LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
  CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
  SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
  INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
  CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
  ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
  POSSIBILITY OF SUCH DAMAGE. */

/* $Id$ */

#ifndef __ERRNO_H_
#define __ERRNO_H_ 1

/** \file */
/** \defgroup avr_errno <errno.h>: System Errors

    \code #include <errno.h>\endcode

    Some functions in the library set the global variable \c errno when an
    error occurs. The file, \c <errno.h>, provides symbolic names for various
    error codes.
 */

#ifdef __cplusplus
extern "C" {
#endif

/** \ingroup avr_errno
    \brief Error code for last error encountered by library

    The variable \c errno holds the last error code encountered by
    a library function.  This variable must be cleared by the
    user prior to calling a library function.

    \warning The \c errno global variable is not safe to use in a threaded or
    multi-task system. A race condition can occur if a task is interrupted
    between the call which sets \c error and when the task examines \c
    errno. If another task changes \c errno during this time, the result will
    be incorrect for the interrupted task. */
extern int errno;

#ifdef __cplusplus
}
#endif

/** \ingroup avr_errno
    \def EDOM

    Domain error. */
#define EDOM       33

/** \ingroup avr_errno
    \def ERANGE

    Range error. */
#define ERANGE     34

#ifndef __DOXYGEN__

/* ((((('E'-64)*26+('N'-64))*26+('O'-64))*26+('S'-64))*26+('Y'-64))*26+'S'-64 */
#define ENOSYS ((int)(66081697 & 0x7fff))

/* (((('E'-64)*26+('I'-64))*26+('N'-64))*26+('T'-64))*26+('R'-64) */
#define EINTR ((int)(2453066 & 0x7fff))

#define E2BIG ENOERR
#define EACCES ENOERR
#define EADDRINUSE ENOERR
#define EADDRNOTAVAIL ENOERR
#define EAFNOSUPPORT ENOERR
```

```
#define EAGAIN ENOERR
#define EALREADY ENOERR
#define EBADF ENOERR
#define EBUSY ENOERR
#define ECHILD ENOERR
#define ECONNABORTED ENOERR
#define ECONNREFUSED ENOERR
#define ECONNRESET ENOERR
#define EDEADLK ENOERR
#define EDESTADDRREQ ENOERR
#define EEXIST ENOERR
#define EFAULT ENOERR
#define EFBIG ENOERR
#define EHOSTUNREACH ENOERR
#define EILSEQ ENOERR
#define EINPROGRESS ENOERR
#define EINVAL ENOERR
#define EIO ENOERR
#define EISCONN ENOERR
#define EISDIR ENOERR
#define ELOOP ENOERR
#define EMFILE ENOERR
#define EMLINK ENOERR
#define EMSGSIZE ENOERR
#define ENAMETOOLONG ENOERR
#define ENETDOWN ENOERR
#define ENETRESET ENOERR
#define ENETUNREACH ENOERR
#define ENFILE ENOERR
#define ENOBUFS ENOERR
#define ENODEV ENOERR
#define ENOENT ENOERR
#define ENOEXEC ENOERR
#define ENOLCK ENOERR
#define ENOMEM ENOERR
#define ENOMSG ENOERR
#define ENOPROTOOPT ENOERR
#define ENOSPC ENOERR
#define ENOTCONN ENOERR
#define ENOTDIR ENOERR
#define ENOTEMPTY ENOERR
#define ENOTSOCK ENOERR
#define ENOTTY ENOERR
#define ENXIO ENOERR
#define EOPNOTSUPP ENOERR
#define EPERM ENOERR
#define EPIPE ENOERR
#define EPROTONOSUPPORT ENOERR
#define EPROTOTYPE ENOERR
#define EROFS ENOERR
#define ESPIPE ENOERR
#define ESRCH ENOERR
#define ETIMEDOUT ENOERR
#define EWOULDBLOCK ENOERR
#define EXDEV ENOERR

/* ((((('E'-64)*26+('N'-64))*26+('O'-64))*26+('E'-64))*26+('R'-64))*26+'R'-64 */
#define ENOERR ((int)(66072050 & 0xffff))

#endif  /* !__DOXYGEN__ */

#endif
```

23.46 inttypes.h File Reference

Macros

macros for printf and scanf format specifiers

For C++, these are only included if __STDC_LIMIT_MACROS is defined before including <inttypes.h>.

- #define PRId8 "d"
- #define PRIdLEAST8 "d"
- #define PRIdFAST8 "d"
- #define PRIi8 "i"
- #define PRIiLEAST8 "i"
- #define PRIiFAST8 "i"
- #define PRId16 "d"

- #define PRIdLEAST16 "d"
- #define PRIdFAST16 "d"
- #define PRIi16 "i"
- #define PRIiLEAST16 "i"
- #define PRIiFAST16 "i"
- #define PRId32 "ld"
- #define PRIdLEAST32 "ld"
- #define PRIdFAST32 "ld"
- #define PRIi32 "li"
- #define PRIiLEAST32 "li"
- #define PRIiFAST32 "li"
- #define PRIdPTR PRId16
- #define PRIiPTR PRIi16
- #define PRIo8 "o"
- #define PRIoLEAST8 "o"
- #define PRIoFAST8 "o"
- #define PRIu8 "u"
- #define PRIuLEAST8 "u"
- #define PRIuFAST8 "u"
- #define PRIx8 "x"
- #define PRIxLEAST8 "x"
- #define PRIxFAST8 "x"
- #define PRIX8 "X"
- #define PRIXLEAST8 "X"
- #define PRIXFAST8 "X"
- #define PRIo16 "o"
- #define PRIoLEAST16 "o"
- #define PRIoFAST16 "o"
- #define PRIu16 "u"
- #define PRIuLEAST16 "u"
- #define PRIuFAST16 "u"
- #define PRIx16 "x"
- #define PRIxLEAST16 "x"
- #define PRIxFAST16 "x"
- #define PRIX16 "X"
- #define PRIXLEAST16 "X"
- #define PRIXFAST16 "X"
- #define PRIo32 "lo"
- #define PRIoLEAST32 "lo"
- #define PRIoFAST32 "lo"
- #define PRIu32 "lu"
- #define PRIuLEAST32 "lu"
- #define PRIuFAST32 "lu"
- #define PRIx32 "lx"
- #define PRIxLEAST32 "lx"
- #define PRIxFAST32 "lx"
- #define PRIX32 "lX"
- #define PRIXLEAST32 "lX"
- #define PRIXFAST32 "lX"
- #define PRIoPTR PRIo16
- #define PRIuPTR PRIu16
- #define PRIxPTR PRIx16
- #define PRIXPTR PRIX16
- #define SCNd8 "hhd"
- #define SCNdLEAST8 "hhd"
- #define SCNdFAST8 "hhd"
- #define SCNi8 "hhi"
- #define SCNiLEAST8 "hhi"
- #define SCNiFAST8 "hhi"
- #define SCNd16 "d"
- #define SCNdLEAST16 "d"
- #define SCNdFAST16 "d"
- #define SCNi16 "i"
- #define SCNiLEAST16 "i"

- #define SCNiFAST16 "i"
- #define SCNd32 "ld"
- #define SCNdLEAST32 "ld"
- #define SCNdFAST32 "ld"
- #define SCNi32 "li"
- #define SCNiLEAST32 "li"
- #define SCNiFAST32 "li"
- #define SCNdPTR SCNd16
- #define SCNiPTR SCNi16
- #define SCNo8 "hho"
- #define SCNoLEAST8 "hho"
- #define SCNoFAST8 "hho"
- #define SCNu8 "hhu"
- #define SCNuLEAST8 "hhu"
- #define SCNuFAST8 "hhu"
- #define SCNx8 "hhx"
- #define SCNxLEAST8 "hhx"
- #define SCNxFAST8 "hhx"
- #define SCNo16 "o"
- #define SCNoLEAST16 "o"
- #define SCNoFAST16 "o"
- #define SCNu16 "u"
- #define SCNuLEAST16 "u"
- #define SCNuFAST16 "u"
- #define SCNx16 "x"
- #define SCNxLEAST16 "x"
- #define SCNxFAST16 "x"
- #define SCNo32 "lo"
- #define SCNoLEAST32 "lo"
- #define SCNoFAST32 "lo"
- #define SCNu32 "lu"
- #define SCNuLEAST32 "lu"
- #define SCNuFAST32 "lu"
- #define SCNx32 "lx"
- #define SCNxLEAST32 "lx"
- #define SCNxFAST32 "lx"
- #define SCNoPTR SCNo16
- #define SCNuPTR SCNu16
- #define SCNxPTR SCNx16

Typedefs

Far pointers for memory access $>$ 64K

- typedef int32_t int_farptr_t
- typedef uint32_t uint_farptr_t

23.47 inttypes.h

Go to the documentation of this file.

```
/* Copyright (c) 2004,2005,2007,2012 Joerg Wunsch
   Copyright (c) 2005, Carlos Lamas
   All rights reserved.

   Redistribution and use in source and binary forms, with or without
   modification, are permitted provided that the following conditions are met:

   * Redistributions of source code must retain the above copyright
     notice, this list of conditions and the following disclaimer.

   * Redistributions in binary form must reproduce the above copyright
     notice, this list of conditions and the following disclaimer in
     the documentation and/or other materials provided with the
     distribution.
```

```

   * Neither the name of the copyright holders nor the names of
     contributors may be used to endorse or promote products derived
     from this software without specific prior written permission.

  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
  AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
  IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
  ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
  LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
  CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
  SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
  INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
  CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
  ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
  POSSIBILITY OF SUCH DAMAGE. */

/* $Id$ */

#ifndef __INTTYPES_H_
#define __INTTYPES_H_

#include <stdint.h>

/** \file */
/** \defgroup avr_inttypes <inttypes.h>: Integer Type conversions
    \code #include <inttypes.h> \endcode

    This header file includes the exact-width integer definitions from
    <tt><stdint.h></tt>, and extends them with additional facilities
    provided by the implementation.

    Currently, the extensions include two additional integer types
    that could hold a "far" pointer (i.e. a code pointer that can
    address more than 64 KB), as well as standard names for all printf
    and scanf formatting options that are supported by the \ref avr_stdio.
    As the library does not support the full range of conversion
    specifiers from ISO 9899:1999, only those conversions that are
    actually implemented will be listed here.

    The idea behind these conversion macros is that, for each of the
    types defined by <stdint.h>, a macro will be supplied that portably
    allows formatting an object of that type in printf() or scanf()
    operations.  Example:

    \code
    #include <inttypes.h>

    uint8_t smallval;
    int32_t longval;
    ...
    printf("The hexadecimal value of smallval is %" PRIx8
           ", the decimal value of longval is %" PRId32 ".\n",
       smallval, longval);
    \endcode
*/

/** \name Far pointers for memory access > 64K */

/**@{*/
/** \ingroup avr_inttypes
    signed integer type that can hold a pointer > 64 KiB */
typedef int32_t int_farptr_t;

/** \ingroup avr_inttypes
    unsigned integer type that can hold a pointer > 64 KiB,
    see also pgm_get_far_address()
 */
typedef uint32_t uint_farptr_t;
/**@}*/

#if !defined(__cplusplus) || defined(__STDC_LIMIT_MACROS)


/** \name macros for printf and scanf format specifiers

    For C++, these are only included if __STDC_LIMIT_MACROS
    is defined before including <inttypes.h>.
 */

/**@{*/
/** \ingroup avr_inttypes
    decimal printf format for int8_t */
#define     PRId8           "d"
/** \ingroup avr_inttypes
    decimal printf format for int_least8_t */
#define     PRIdLEAST8      "d"
```

```
/** \ingroup avr_inttypes
    decimal printf format for int_fast8_t */
#define     PRIdFAST8       "d"

/** \ingroup avr_inttypes
    integer printf format for int8_t */
#define     PRIi8           "i"
/** \ingroup avr_inttypes
    integer printf format for int_least8_t */
#define     PRIiLEAST8      "i"
/** \ingroup avr_inttypes
    integer printf format for int_fast8_t */
#define     PRIiFAST8       "i"


/** \ingroup avr_inttypes
    decimal printf format for int16_t */
#define     PRId16          "d"
/** \ingroup avr_inttypes
    decimal printf format for int_least16_t */
#define     PRIdLEAST16     "d"
/** \ingroup avr_inttypes
    decimal printf format for int_fast16_t */
#define     PRIdFAST16      "d"

/** \ingroup avr_inttypes
    integer printf format for int16_t */
#define     PRIi16          "i"
/** \ingroup avr_inttypes
    integer printf format for int_least16_t */
#define     PRIiLEAST16     "i"
/** \ingroup avr_inttypes
    integer printf format for int_fast16_t */
#define     PRIiFAST16      "i"


/** \ingroup avr_inttypes
    decimal printf format for int32_t */
#define     PRId32          "ld"
/** \ingroup avr_inttypes
    decimal printf format for int_least32_t */
#define     PRIdLEAST32     "ld"
/** \ingroup avr_inttypes
    decimal printf format for int_fast32_t */
#define     PRIdFAST32      "ld"

/** \ingroup avr_inttypes
    integer printf format for int32_t */
#define     PRIi32          "li"
/** \ingroup avr_inttypes
    integer printf format for int_least32_t */
#define     PRIiLEAST32     "li"
/** \ingroup avr_inttypes
    integer printf format for int_fast32_t */
#define     PRIiFAST32      "li"


#ifdef __avr_libc_does_not_implement_long_long_in_printf_or_scanf

#define     PRId64          "lld"
#define     PRIdLEAST64     "lld"
#define     PRIdFAST64      "lld"

#define     PRIi64          "lli"
#define     PRIiLEAST64     "lli"
#define     PRIiFAST64      "lli"


#define     PRIdMAX         "lld"
#define     PRIiMAX         "lli"

#endif

/** \ingroup avr_inttypes
    decimal printf format for intptr_t */
#define     PRIdPTR         PRId16
/** \ingroup avr_inttypes
    integer printf format for intptr_t */
#define     PRIiPTR         PRIi16

/** \ingroup avr_inttypes
    octal printf format for uint8_t */
#define     PRIo8           "o"
/** \ingroup avr_inttypes
    octal printf format for uint_least8_t */
#define     PRIoLEAST8      "o"
/** \ingroup avr_inttypes
```

```
    octal printf format for uint_fast8_t */
#define     PRIoFAST8       "o"

/** \ingroup avr_inttypes
    decimal printf format for uint8_t */
#define     PRIu8           "u"
/** \ingroup avr_inttypes
    decimal printf format for uint_least8_t */
#define     PRIuLEAST8      "u"
/** \ingroup avr_inttypes
    decimal printf format for uint_fast8_t */
#define     PRIuFAST8       "u"

/** \ingroup avr_inttypes
    hexadecimal printf format for uint8_t */
#define     PRIx8           "x"
/** \ingroup avr_inttypes
    hexadecimal printf format for uint_least8_t */
#define     PRIxLEAST8      "x"
/** \ingroup avr_inttypes
    hexadecimal printf format for uint_fast8_t */
#define     PRIxFAST8       "x"

/** \ingroup avr_inttypes
    uppercase hexadecimal printf format for uint8_t */
#define     PRIX8           "X"
/** \ingroup avr_inttypes
    uppercase hexadecimal printf format for uint_least8_t */
#define     PRIXLEAST8      "X"
/** \ingroup avr_inttypes
    uppercase hexadecimal printf format for uint_fast8_t */
#define     PRIXFAST8       "X"


/** \ingroup avr_inttypes
    octal printf format for uint16_t */
#define     PRIo16          "o"
/** \ingroup avr_inttypes
    octal printf format for uint_least16_t */
#define     PRIoLEAST16     "o"
/** \ingroup avr_inttypes
    octal printf format for uint_fast16_t */
#define     PRIoFAST16      "o"

/** \ingroup avr_inttypes
    decimal printf format for uint16_t */
#define     PRIu16          "u"
/** \ingroup avr_inttypes
    decimal printf format for uint_least16_t */
#define     PRIuLEAST16     "u"
/** \ingroup avr_inttypes
    decimal printf format for uint_fast16_t */
#define     PRIuFAST16      "u"

/** \ingroup avr_inttypes
    hexadecimal printf format for uint16_t */
#define     PRIx16          "x"
/** \ingroup avr_inttypes
    hexadecimal printf format for uint_least16_t */
#define     PRIxLEAST16     "x"
/** \ingroup avr_inttypes
    hexadecimal printf format for uint_fast16_t */
#define     PRIxFAST16      "x"

/** \ingroup avr_inttypes
    uppercase hexadecimal printf format for uint16_t */
#define     PRIX16          "X"
/** \ingroup avr_inttypes
    uppercase hexadecimal printf format for uint_least16_t */
#define     PRIXLEAST16     "X"
/** \ingroup avr_inttypes
    uppercase hexadecimal printf format for uint_fast16_t */
#define     PRIXFAST16      "X"


/** \ingroup avr_inttypes
    octal printf format for uint32_t */
#define     PRIo32          "lo"
/** \ingroup avr_inttypes
    octal printf format for uint_least32_t */
#define     PRIoLEAST32     "lo"
/** \ingroup avr_inttypes
    octal printf format for uint_fast32_t */
#define     PRIoFAST32      "lo"

/** \ingroup avr_inttypes
    decimal printf format for uint32_t */
```

```
#define     PRIu32          "lu"
/** \ingroup avr_inttypes
    decimal printf format for uint_least32_t */
#define     PRIuLEAST32     "lu"
/** \ingroup avr_inttypes
    decimal printf format for uint_fast32_t */
#define     PRIuFAST32      "lu"

/** \ingroup avr_inttypes
    hexadecimal printf format for uint32_t */
#define     PRIx32          "lx"
/** \ingroup avr_inttypes
    hexadecimal printf format for uint_least32_t */
#define     PRIxLEAST32     "lx"
/** \ingroup avr_inttypes
    hexadecimal printf format for uint_fast32_t */
#define     PRIxFAST32      "lx"

/** \ingroup avr_inttypes
    uppercase hexadecimal printf format for uint32_t */
#define     PRIX32          "lX"
/** \ingroup avr_inttypes
    uppercase hexadecimal printf format for uint_least32_t */
#define     PRIXLEAST32     "lX"
/** \ingroup avr_inttypes
    uppercase hexadecimal printf format for uint_fast32_t */
#define     PRIXFAST32      "lX"


#ifdef __avr_libc_does_not_implement_long_long_in_printf_or_scanf

#define     PRIo64          "llo"
#define     PRIoLEAST64     "llo"
#define     PRIoFAST64      "llo"

#define     PRIu64          "llu"
#define     PRIuLEAST64     "llu"
#define     PRIuFAST64      "llu"

#define     PRIx64          "llx"
#define     PRIxLEAST64     "llx"
#define     PRIxFAST64      "llx"

#define     PRIX64          "llX"
#define     PRIXLEAST64     "llX"
#define     PRIXFAST64      "llX"

#define     PRIoMAX         "llo"
#define     PRIuMAX         "llu"
#define     PRIxMAX         "llx"
#define     PRIXMAX         "llX"

#endif

/** \ingroup avr_inttypes
    octal printf format for uintptr_t */
#define     PRIoPTR         PRIo16
/** \ingroup avr_inttypes
    decimal printf format for uintptr_t */
#define     PRIuPTR         PRIu16
/** \ingroup avr_inttypes
    hexadecimal printf format for uintptr_t */
#define     PRIxPTR         PRIx16
/** \ingroup avr_inttypes
    uppercase hexadecimal printf format for uintptr_t */
#define     PRIXPTR         PRIX16


/** \ingroup avr_inttypes
    decimal scanf format for int8_t */
#define     SCNd8           "hhd"
/** \ingroup avr_inttypes
    decimal scanf format for int_least8_t */
#define     SCNdLEAST8      "hhd"
/** \ingroup avr_inttypes
    decimal scanf format for int_fast8_t */
#define     SCNdFAST8       "hhd"

/** \ingroup avr_inttypes
    generic-integer scanf format for int8_t */
#define     SCNi8           "hhi"
/** \ingroup avr_inttypes
    generic-integer scanf format for int_least8_t */
#define     SCNiLEAST8      "hhi"
/** \ingroup avr_inttypes
    generic-integer scanf format for int_fast8_t */
#define     SCNiFAST8       "hhi"
```

```


/** \ingroup avr_inttypes
    decimal scanf format for int16_t */
#define     SCNd16          "d"
/** \ingroup avr_inttypes
    decimal scanf format for int_least16_t */
#define     SCNdLEAST16     "d"
/** \ingroup avr_inttypes
    decimal scanf format for int_fast16_t */
#define     SCNdFAST16      "d"

/** \ingroup avr_inttypes
    generic-integer scanf format for int16_t */
#define     SCNi16          "i"
/** \ingroup avr_inttypes
    generic-integer scanf format for int_least16_t */
#define     SCNiLEAST16     "i"
/** \ingroup avr_inttypes
    generic-integer scanf format for int_fast16_t */
#define     SCNiFAST16      "i"


/** \ingroup avr_inttypes
    decimal scanf format for int32_t */
#define     SCNd32          "ld"
/** \ingroup avr_inttypes
    decimal scanf format for int_least32_t */
#define     SCNdLEAST32     "ld"
/** \ingroup avr_inttypes
    decimal scanf format for int_fast32_t */
#define     SCNdFAST32      "ld"

/** \ingroup avr_inttypes
    generic-integer scanf format for int32_t */
#define     SCNi32          "li"
/** \ingroup avr_inttypes
    generic-integer scanf format for int_least32_t */
#define     SCNiLEAST32     "li"
/** \ingroup avr_inttypes
    generic-integer scanf format for int_fast32_t */
#define     SCNiFAST32      "li"


#ifdef __avr_libc_does_not_implement_long_long_in_printf_or_scanf

#define     SCNd64          "lld"
#define     SCNdLEAST64     "lld"
#define     SCNdFAST64      "lld"

#define     SCNi64          "lli"
#define     SCNiLEAST64     "lli"
#define     SCNiFAST64      "lli"

#define     SCNdMAX         "lld"
#define     SCNiMAX         "lli"

#endif

/** \ingroup avr_inttypes
    decimal scanf format for intptr_t */
#define     SCNdPTR         SCNd16
/** \ingroup avr_inttypes
    generic-integer scanf format for intptr_t */
#define     SCNiPTR         SCNi16

/** \ingroup avr_inttypes
    octal scanf format for uint8_t */
#define     SCNo8           "hho"
/** \ingroup avr_inttypes
    octal scanf format for uint_least8_t */
#define     SCNoLEAST8      "hho"
/** \ingroup avr_inttypes
    octal scanf format for uint_fast8_t */
#define     SCNoFAST8       "hho"

/** \ingroup avr_inttypes
    decimal scanf format for uint8_t */
#define     SCNu8           "hhu"
/** \ingroup avr_inttypes
    decimal scanf format for uint_least8_t */
#define     SCNuLEAST8      "hhu"
/** \ingroup avr_inttypes
    decimal scanf format for uint_fast8_t */
#define     SCNuFAST8       "hhu"

/** \ingroup avr_inttypes
```

```
    hexadecimal scanf format for uint8_t */
#define     SCNx8           "hhx"
/** \ingroup avr_inttypes
    hexadecimal scanf format for uint_least8_t */
#define     SCNxLEAST8      "hhx"
/** \ingroup avr_inttypes
    hexadecimal scanf format for uint_fast8_t */
#define     SCNxFAST8       "hhx"

/** \ingroup avr_inttypes
    octal scanf format for uint16_t */
#define     SCNo16          "o"
/** \ingroup avr_inttypes
    octal scanf format for uint_least16_t */
#define     SCNoLEAST16     "o"
/** \ingroup avr_inttypes
    octal scanf format for uint_fast16_t */
#define     SCNoFAST16      "o"

/** \ingroup avr_inttypes
    decimal scanf format for uint16_t */
#define     SCNu16          "u"
/** \ingroup avr_inttypes
    decimal scanf format for uint_least16_t */
#define     SCNuLEAST16     "u"
/** \ingroup avr_inttypes
    decimal scanf format for uint_fast16_t */
#define     SCNuFAST16      "u"

/** \ingroup avr_inttypes
    hexadecimal scanf format for uint16_t */
#define     SCNx16          "x"
/** \ingroup avr_inttypes
    hexadecimal scanf format for uint_least16_t */
#define     SCNxLEAST16     "x"
/** \ingroup avr_inttypes
    hexadecimal scanf format for uint_fast16_t */
#define     SCNxFAST16      "x"


/** \ingroup avr_inttypes
    octal scanf format for uint32_t */
#define     SCNo32          "lo"
/** \ingroup avr_inttypes
    octal scanf format for uint_least32_t */
#define     SCNoLEAST32     "lo"
/** \ingroup avr_inttypes
    octal scanf format for uint_fast32_t */
#define     SCNoFAST32      "lo"

/** \ingroup avr_inttypes
    decimal scanf format for uint32_t */
#define     SCNu32          "lu"
/** \ingroup avr_inttypes
    decimal scanf format for uint_least32_t */
#define     SCNuLEAST32     "lu"
/** \ingroup avr_inttypes
    decimal scanf format for uint_fast32_t */
#define     SCNuFAST32      "lu"

/** \ingroup avr_inttypes
    hexadecimal scanf format for uint32_t */
#define     SCNx32          "lx"
/** \ingroup avr_inttypes
    hexadecimal scanf format for uint_least32_t */
#define     SCNxLEAST32     "lx"
/** \ingroup avr_inttypes
    hexadecimal scanf format for uint_fast32_t */
#define     SCNxFAST32      "lx"


#ifdef __avr_libc_does_not_implement_long_long_in_printf_or_scanf

#define     SCNo64          "llo"
#define     SCNoLEAST64     "llo"
#define     SCNoFAST64      "llo"

#define     SCNu64          "llu"
#define     SCNuLEAST64     "llu"
#define     SCNuFAST64      "llu"

#define     SCNx64          "llx"
#define     SCNxLEAST64     "llx"
#define     SCNxFAST64      "llx"

#define     SCNoMAX         "llo"
#define     SCNuMAX         "llu"
```

```
#define     SCNxMAX         "llx"

#endif

/** \ingroup avr_inttypes
    octal scanf format for uintptr_t */
#define     SCNoPTR         SCNo16
/** \ingroup avr_inttypes
    decimal scanf format for uintptr_t */
#define     SCNuPTR         SCNu16
/** \ingroup avr_inttypes
    hexadecimal scanf format for uintptr_t */
#define     SCNxPTR         SCNx16

/**@}*/


#endif  /* !defined(__cplusplus) || defined(__STDC_LIMIT_MACROS) */


#endif /* __INTTYPES_H_ */
```

23.48 math.h File Reference

Macros

- #define M_E 2.7182818284590452354
- #define M_LOG2E 1.4426950408889634074
- #define M_LOG10E 0.43429448190325182765
- #define M_LN2 0.69314718055994530942
- #define M_LN10 2.30258509299404568402
- #define M_PI 3.14159265358979323846
- #define M_PI_2 1.57079632679489661923
- #define M_PI_4 0.78539816339744830962
- #define M_1_PI 0.31830988618379067154
- #define M_2_PI 0.63661977236758134308
- #define M_2_SQRTPI 1.12837916709551257390
- #define M_SQRT2 1.41421356237309504880
- #define M_SQRT1_2 0.70710678118654752440
- #define NAN __builtin_nan("")
- #define nanf(__tagp) __builtin_nanf(__tag)
- #define nan(__tag) __builtin_nan(__tag)
- #define nanl(__tag) __builtin_nanl(__tag)
- #define INFINITY __builtin_inf()
- #define HUGE_VALF __builtin_huge_valf()
- #define HUGE_VAL __builtin_huge_val()
- #define HUGE_VALL __builtin_huge_vall()

Functions

- float cosf (float x)
- double cos (double x)
- long double cosl (long double x)
- float sinf (float x)
- double sin (double x)
- long double sinl (long double x)
- float tanf (float x)
- double tan (double x)
- long double tanl (long double x)
- static float fabsf (float __x)

- static double fabs (double __x)
- static long double fabsl (long double __x)
- float fmodf (float x, float y)
- double fmod (double x, double y)
- long double fmodl (long double x, long double y)
- float modff (float x, float ∗iptr)
- double modf (double x, double ∗iptr)
- long double modfl (long double x, long double ∗iptr)
- float sqrtf (float x)
- double sqrt (double x)
- long double sqrtl (long double x)
- float cbrtf (float x)
- double cbrt (double x)
- long double cbrtl (long double x)
- float hypotf (float x, float y)
- double hypot (double x, double y)
- long double hypotl (long double x, long double y)
- float floorf (float x)
- double floor (double x)
- long double floorl (long double x)
- float ceilf (float x)
- double ceil (double x)
- long double ceill (long double x)
- float frexpf (float x, int ∗pexp)
- double frexp (double x, int ∗pexp)
- long double frexpl (long double x, int ∗pexp)
- float ldexpf (float x, int iexp)
- double ldexp (double x, int iexp)
- long double ldexpl (long double x, int iexp)
- float expf (float x)
- double exp (double x)
- long double expl (long double x)
- float coshf (float x)
- double cosh (double x)
- long double coshl (long double x)
- float sinhf (float x)
- double sinh (double x)
- long double sinhl (long double x)
- float tanhf (float x)
- double tanh (double x)
- long double tanhl (long double x)
- float acosf (float x)
- double acos (double x)
- long double acosl (long double x)
- float asinf (float x)
- double asin (double x)
- long double asinl (long double x)
- float atanf (float x)
- double atan (double x)
- long double atanl (long double x)
- float atan2f (float y, float x)
- double atan2 (double y, double x)
- long double atan2l (long double y, long double x)
- float logf (float x)
- double log (double x)

- long double logl (long double x)
- float log10f (float x)
- double log10 (double x)
- long double log10l (long double x)
- float powf (float x, float y)
- double pow (double x, double y)
- long double powl (long double x, long double y)
- int isnanf (float x)
- int isnan (double x)
- int isnanl (long double x)
- int isinff (float x)
- int isinf (double x)
- int isinfl (long double x)
- static int isfinitef (float __x)
- static int isfinite (double __x)
- static int isfinitel (long double __x)
- static float copysignf (float __x, float __y)
- static double copysign (double __x, double __y)
- static long double copysignl (long double __x, long double __y)
- int signbitf (float x)
- int signbit (double x)
- int signbitl (long double x)
- float fdimf (float x, float y)
- double fdim (double x, double y)
- long double fdiml (long double x, long double y)
- float fmaf (float x, float y, float z)
- double fma (double x, double y, double z)
- long double fmal (long double x, long double y, long double z)
- float fmaxf (float x, float y)
- double fmax (double x, double y)
- long double fmaxl (long double x, long double y)
- float fminf (float x, float y)
- double fmin (double x, double y)
- long double fminl (long double x, long double y)
- float truncf (float x)
- double trunc (double x)
- long double truncl (long double x)
- float roundf (float x)
- double round (double x)
- long double roundl (long double x)
- long lroundf (float x)
- long lround (double x)
- long lroundl (long double x)
- long lrintf (float x)
- long lrint (double x)
- long lrintl (long double x)

Non-Standard Math Functions

- float squaref (float x)
- double square (double x)
- long double squarel (long double x)

23.49 math.h

Go to the documentation of this file.

```
/* Copyright (c) 2002,2007-2009 Michael Stumpf

   Portions of documentation Copyright (c) 1990 - 1994
   The Regents of the University of California.

   All rights reserved.

   Redistribution and use in source and binary forms, with or without
   modification, are permitted provided that the following conditions are met:

   * Redistributions of source code must retain the above copyright
     notice, this list of conditions and the following disclaimer.

   * Redistributions in binary form must reproduce the above copyright
     notice, this list of conditions and the following disclaimer in
     the documentation and/or other materials provided with the
     distribution.

   * Neither the name of the copyright holders nor the names of
     contributors may be used to endorse or promote products derived
     from this software without specific prior written permission.

  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
  AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
  IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
  ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
  LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
  CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
  SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
  INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
  CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
  ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
  POSSIBILITY OF SUCH DAMAGE. */

/* $Id$ */

/*
   math.h - mathematical functions

   Author : Michael Stumpf
            Michael.Stumpf@t-online.de

   __ATTR_CONST__ added by marekm@linux.org.pl for functions
   that "do not examine any values except their arguments, and have
   no effects except the return value", for better optimization by gcc.
 */

#ifndef __MATH_H
#define __MATH_H

#ifdef __cplusplus
extern "C" {
#endif

/** \file */
/** \defgroup avr_math <math.h>: Mathematics
    \code #include <math.h> \endcode

    This header file declares basic mathematics constants and
    functions.

    \par Notes:
    - Math functions do not raise exceptions and do not change the
      \c errno variable. Therefore the majority of them are declared
      with \c const attribute, for better optimization by GCC.
    - 64-bit floating-point arithmetic is only available in
      <a href="https://gcc.gnu.org/gcc-10/changes.html#avr">avr-gcc v10</a>
      and up.
      The size of the \c double and \c long \c double type can be selected
      at compile-time with options like <tt>-mdouble=64</tt> and
      <tt>-mlong-double=32</tt>.  Whether such options are available,
      and their default values,
      depend on how the compiler has been configured.
    - The implementation of 64-bit floating-point arithmetic has some
      shortcomings and limitations, see the
      <a href="https://gcc.gnu.org/wiki/avr-gcc#Libf7">avr-gcc Wiki</a>
      for details.
    - In order to access the <tt>float</tt> functions,
      in avr-gcc v4.6 and older it is usually
      also required to link with \c -lm. In avr-gcc v4.7 and up, \c -lm
      is added automatically to all linker invocations.
*/

```

```

/** \ingroup avr_math */
/**@{*/

/** The constant Euler's number \a e. */
#define M_E 2.7182818284590452354

/** The constant logarithm of Euler's number \a e to base 2. */
#define M_LOG2E 1.4426950408889634074

/** The constant logarithm of Euler's number \a e to base 10. */
#define M_LOG10E    0.43429448190325182765

/** The constant natural logarithm of 2.    */
#define M_LN2       0.69314718055994530942

/** The constant natural logarithm of 10.   */
#define M_LN10      2.30258509299404568402

/** The constant \a pi. */
#define M_PI        3.14159265358979323846

/** The constant \a pi/2.   */
#define M_PI_2      1.57079632679489661923

/** The constant \a pi/4.   */
#define M_PI_4      0.78539816339744830962

/** The constant \a 1/pi.   */
#define M_1_PI      0.31830988618379067154

/** The constant \a 2/pi.   */
#define M_2_PI      0.63661977236758134308

/** The constant \a 2/sqrt(pi). */
#define M_2_SQRTPI  1.12837916709551257390

/** The square root of 2.   */
#define M_SQRT2     1.41421356237309504880

/** The constant \a 1/sqrt(2).  */
#define M_SQRT1_2   0.70710678118654752440

/** The \c double representation of a constant quiet NaN. */
#define NAN __builtin_nan("")

/** The \c float representation of a constant quiet NaN.
    \p __tag is a string constant like \c "" or \c "123". */
#define nanf(__tagp)    __builtin_nanf(__tag)

/** The \c double representation of a constant quiet NaN.
    \p __tag is a string constant like \c "" or \c "123". */
#define nan(__tag)  __builtin_nan(__tag)

/** The \c long \c double representation of a constant quiet NaN.
    \p __tag is a string constant like \c "" or \c "123". */
#define nanl(__tag) __builtin_nanl(__tag)

/** \c double infinity constant.    */
#define INFINITY __builtin_inf()

/** \c float infinity constant. */
#define HUGE_VALF __builtin_huge_valf()

/** \c double infinity constant.    */
#define HUGE_VAL __builtin_huge_val()

/** \c long \c double infinity constant.    */
#define HUGE_VALL __builtin_huge_vall()

#ifndef __DOXYGEN__
#ifndef  __ATTR_CONST__
# define __ATTR_CONST__ __attribute__((__const__))
#endif

#ifndef __ATTR_ALWAYS_INLINE__
#define __ATTR_ALWAYS_INLINE__ __inline__ __attribute__((__always_inline__))
#endif
#endif /* ! DOXYGEN */

/** The cosf() function returns the cosine of \a x, measured in radians. */
__ATTR_CONST__ extern float cosf (float x);
/** The cos() function returns the cosine of \a x, measured in radians. */
__ATTR_CONST__ extern double cos (double x);
/** The cosl() function returns the cosine of \a x, measured in radians. */
__ATTR_CONST__ extern long double cosl (long double x);

```

```c
/** The sinf() function returns the sine of \a x, measured in radians. */
__ATTR_CONST__ extern float sinf (float x);
/** The sin() function returns the sine of \a x, measured in radians. */
__ATTR_CONST__ extern double sin (double x);
/** The sinl() function returns the sine of \a x, measured in radians. */
__ATTR_CONST__ extern long double sinl (long double x);

/** The tanf() function returns the tangent of \a x, measured in radians. */
__ATTR_CONST__ extern float tanf (float x);
/** The tan() function returns the tangent of \a x, measured in radians. */
__ATTR_CONST__ extern double tan (double x);
/** The tanl() function returns the tangent of \a x, measured in radians. */
__ATTR_CONST__ extern long double tanl (long double x);

/** The fabsf() function computes the absolute value of a floating-point number \a x. */
static __ATTR_ALWAYS_INLINE__ float fabsf (float __x)
{
    return __builtin_fabsf (__x);
}

/** The fabs() function computes the absolute value of a floating-point number \a x. */
static __ATTR_ALWAYS_INLINE__ double fabs (double __x)
{
    return __builtin_fabs (__x);
}

/** The fabsl() function computes the absolute value of a floating-point number \a x. */
static __ATTR_ALWAYS_INLINE__ long double fabsl (long double __x)
{
    return __builtin_fabsl (__x);
}

/** The function fmodf() returns the floating-point remainder of <em>x / y</em>. */
__ATTR_CONST__ extern float fmodf (float x, float y);
/** The function fmod() returns the floating-point remainder of <em>x / y</em>. */
__ATTR_CONST__ extern double fmod (double x, double y);
/** The function fmodl() returns the floating-point remainder of <em>x / y</em>. */
__ATTR_CONST__ extern long double fmodl (long double x, long double y);

/** The modff() function breaks the argument \a x into integral and
    fractional parts, each of which has the same sign as the argument.
    It stores the integral part as a \c float in the object pointed to by
    \a iptr.

    The modff() function returns the signed fractional part of \a x.

    \note This implementation skips writing by zero pointer.  However,
    the GCC 4.3 can replace this function with inline code that does not
    permit to use NULL address for the avoiding of storing. */
extern float modff (float x, float *iptr);
/** The modf() function breaks the argument \a x into integral and
    fractional parts, each of which has the same sign as the argument.
    It stores the integral part as a \c double in the object pointed to by
    \a iptr.

    The modf() function returns the signed fractional part of \a x. */
extern double modf (double x, double *iptr);
/** The modfl() function breaks the argument \a x into integral and
    fractional parts, each of which has the same sign as the argument.
    It stores the integral part as a \c long \c double in the object pointed to by
    \a iptr.

    The modf() function returns the signed fractional part of \a x. */
extern long double modfl (long double x, long double *iptr);

/** The sqrtf() function returns the non-negative square root of \a x. */
__ATTR_CONST__ extern float sqrtf (float x);
/** The sqrt() function returns the non-negative square root of \a x. */
__ATTR_CONST__ extern double sqrt (double x);
/** The sqrtl() function returns the non-negative square root of \a x. */
__ATTR_CONST__ extern long double sqrtl (long double x);

/** The cbrtf() function returns the cube root of \a x. */
__ATTR_CONST__ extern float cbrtf (float x);
/** The cbrt() function returns the cube root of \a x. */
__ATTR_CONST__ extern double cbrt (double x);
/** The cbrtl() function returns the cube root of \a x. */
__ATTR_CONST__ extern long double cbrtl (long double x);

/** The hypotf() function returns <em>sqrtf(x*x + y*y)</em>. This
    is the length of the hypotenuse of a right triangle with sides of
    length \a x and \a y, or the  distance of the point (\a x, \a
    y) from the origin. Using this function  instead of the direct
    formula is wise, since the error is much smaller. No underflow with
    small \a x and \a y. No overflow if result is in range. */
__ATTR_CONST__ extern float hypotf (float x, float y);
/** The hypot() function returns <em>sqrt(x*x + y*y)</em>. This
```

```
    is the length of the hypotenuse of a right triangle with sides of
    length \a x and \a y, or the  distance of the point (\a x, \a
    y) from the origin. Using this function  instead of the direct
    formula is wise, since the error is much smaller. No underflow with
    small \a x and \a y. No overflow if result is in range. */
__ATTR_CONST__ extern double hypot (double x, double y);
/** The hypotl() function returns <em>sqrtl(x*x + y*y)</em>. This
    is the length of the hypotenuse of a right triangle with sides of
    length \a x and \a y, or the  distance of the point (\a x, \a
    y) from the origin. Using this function  instead of the direct
    formula is wise, since the error is much smaller. No underflow with
    small \a x and \a y. No overflow if result is in range. */
__ATTR_CONST__ extern long double hypotl (long double x, long double y);

/** The floorf() function returns the largest integral value less than or
    equal to \a x, expressed as a floating-point number. */
__ATTR_CONST__ extern float floorf (float x);
/** The floor() function returns the largest integral value less than or
    equal to \a x, expressed as a floating-point number. */
__ATTR_CONST__ extern double floor (double x);
/** The floorl() function returns the largest integral value less than or
    equal to \a x, expressed as a floating-point number. */
__ATTR_CONST__ extern long double floorl (long double x);

/** The ceilf() function returns the smallest integral value greater than
    or equal to \a x, expressed as a floating-point number. */
__ATTR_CONST__ extern float ceilf (float x);
/** The ceil() function returns the smallest integral value greater than
    or equal to \a x, expressed as a floating-point number. */
__ATTR_CONST__ extern double ceil (double x);
/** The ceill() function returns the smallest integral value greater than
    or equal to \a x, expressed as a floating-point number. */
__ATTR_CONST__ extern long double ceill (long double x);

/** The frexpf() function breaks a floating-point number into a normalized
    fraction and an integral power of 2.  It stores the integer in the \c
    int object pointed to by \a pexp.

    If \a x is a normal float point number, the frexpf() function
    returns the value \c v, such that \c v has a magnitude in the
    interval [1/2, 1) or zero, and \a x equals \c v times 2 raised to
    the power \a pexp. If \a x is zero, both parts of the result are
    zero. If \a x is not a finite number, the frexpf() returns \a x as
    is and stores 0 by \a pexp.

    \note  This implementation permits a zero pointer as a directive to
    skip a storing the exponent.
 */
extern float frexpf (float x, int *pexp);
/** The frexp() function breaks a floating-point number into a normalized
    fraction and an integral power of 2.  It stores the integer in the \c
    int object pointed to by \a pexp.

    If \a x is a normal float point number, the frexp() function
    returns the value \c v, such that \c v has a magnitude in the
    interval [1/2, 1) or zero, and \a x equals \c v times 2 raised to
    the power \a pexp. If \a x is zero, both parts of the result are
    zero. If \a x is not a finite number, the frexp() returns \a x as
    is and stores 0 by \a pexp. */
extern double frexp (double x, int *pexp);
/** The frexpl() function breaks a floating-point number into a normalized
    fraction and an integral power of 2.  It stores the integer in the \c
    int object pointed to by \a pexp.

    If \a x is a normal float point number, the frexpl() function
    returns the value \c v, such that \c v has a magnitude in the
    interval [1/2, 1) or zero, and \a x equals \c v times 2 raised to
    the power \a pexp. If \a x is zero, both parts of the result are
    zero. If \a x is not a finite number, the frexpl() returns \a x as
    is and stores 0 by \a pexp. */
extern long double frexpl (long double x, int *pexp);

/** The ldexpf() function multiplies a floating-point number by an integral
    power of 2. It returns the value of \a x times 2 raised to the power
    \a iexp. */
__ATTR_CONST__ extern float ldexpf (float x, int iexp);
/** The ldexp() function multiplies a floating-point number by an integral
    power of 2. It returns the value of \a x times 2 raised to the power
    \a iexp. */
__ATTR_CONST__ extern double ldexp (double x, int iexp);
/** The ldexpl() function multiplies a floating-point number by an integral
    power of 2. It returns the value of \a x times 2 raised to the power
    \a iexp. */
__ATTR_CONST__ extern long double ldexpl (long double x, int iexp);

/** The expf() function returns the exponential value of \a x. */
__ATTR_CONST__ extern float expf (float x);
```

```
/** The exp() function returns the exponential value of \a x. */
__ATTR_CONST__ extern double exp (double x);
/** The expl() function returns the exponential value of \a x. */
__ATTR_CONST__ extern long double expl (long double x);

/** The coshf() function returns the hyperbolic cosine of \a x. */
__ATTR_CONST__ extern float coshf (float x);
/** The cosh() function returns the hyperbolic cosine of \a x. */
__ATTR_CONST__ extern double cosh (double x);
/** The coshl() function returns the hyperbolic cosine of \a x. */
__ATTR_CONST__ extern long double coshl (long double x);

/** The sinhf() function returns the hyperbolic sine of \a x. */
__ATTR_CONST__ extern float sinhf (float x);
/** The sinh() function returns the hyperbolic sine of \a x. */
__ATTR_CONST__ extern double sinh (double x);
/** The sinhl() function returns the hyperbolic sine of \a x. */
__ATTR_CONST__ extern long double sinhl (long double x);

/** The tanhf() function returns the hyperbolic tangent of \a x. */
__ATTR_CONST__ extern float tanhf (float x);
/** The tanh() function returns the hyperbolic tangent of \a x. */
__ATTR_CONST__ extern double tanh (double x);
/** The tanhl() function returns the hyperbolic tangent of \a x. */
__ATTR_CONST__ extern long double tanhl (long double x);

/** The acosf() function computes the principal value of the arc cosine of
    \a x.  The returned value is in the range [0, pi] radians. A domain
    error occurs for arguments not in the range [&minus;1, +1]. */
__ATTR_CONST__ extern float acosf (float x);
/** The acos() function computes the principal value of the arc cosine of
    \a x.  The returned value is in the range [0, pi] radians or NaN. */
__ATTR_CONST__ extern double acos (double x);
/** The acosl() function computes the principal value of the arc cosine of
    \a x.  The returned value is in the range [0, pi] radians or NaN. */
__ATTR_CONST__ extern long double acosl (long double x);

/** The asinf() function computes the principal value of the arc sine of
    \a x.  The returned value is in the range [&minus;pi/2, pi/2] radians. A
    domain error occurs for arguments not in the range [&minus;1, +1]. */
__ATTR_CONST__ extern float asinf (float x);
/** The asin() function computes the principal value of the arc sine of
    \a x.  The returned value is in the range [&minus;pi/2, pi/2] radians or NaN. */
__ATTR_CONST__ extern double asin (double x);
/** The asinl() function computes the principal value of the arc sine of
    \a x.  The returned value is in the range [&minus;pi/2, pi/2] radians or NaN. */
__ATTR_CONST__ extern long double asinl (long double x);

/** The atanf() function computes the principal value of the arc tangent
    of \a x.  The returned value is in the range [&minus;pi/2, pi/2] radians. */
__ATTR_CONST__ extern float atanf (float x);
/** The atan() function computes the principal value of the arc tangent
    of \a x.  The returned value is in the range [&minus;pi/2, pi/2] radians. */
__ATTR_CONST__ extern double atan (double x);
/** The atanl() function computes the principal value of the arc tangent
    of \a x.  The returned value is in the range [&minus;pi/2, pi/2] radians. */
__ATTR_CONST__ extern long double atanl (long double x);

/** The atan2f() function computes the principal value of the arc tangent
    of <em>y / x</em>, using the signs of both arguments to determine
    the quadrant of the return value.  The returned value is in the range
    [&minus;pi, +pi] radians. */
__ATTR_CONST__ extern float atan2f (float y, float x);
/** The atan2() function computes the principal value of the arc tangent
    of <em>y / x</em>, using the signs of both arguments to determine
    the quadrant of the return value.  The returned value is in the range
    [&minus;pi, +pi] radians. */
__ATTR_CONST__ extern double atan2 (double y, double x);
/** The atan2l() function computes the principal value of the arc tangent
    of <em>y / x</em>, using the signs of both arguments to determine
    the quadrant of the return value.  The returned value is in the range
    [&minus;pi, +pi] radians. */
__ATTR_CONST__ extern long double atan2l (long double y, long double x);

/** The logf() function returns the natural logarithm of argument \a x. */
__ATTR_CONST__ extern float logf (float x);
/** The log() function returns the natural logarithm of argument \a x. */
__ATTR_CONST__ extern double log (double x);
/** The logl() function returns the natural logarithm of argument \a x. */
__ATTR_CONST__ extern long double logl (long double x);

/** The log10f() function returns the logarithm of argument \a x to base 10. */
__ATTR_CONST__ extern float log10f (float x);
/** The log10() function returns the logarithm of argument \a x to base 10. */
__ATTR_CONST__ extern double log10 (double x);
/** The log10l() function returns the logarithm of argument \a x to base 10. */
__ATTR_CONST__ extern long double log10l (long double x);
```

```
/** The function powf() returns the value of \a x to the exponent \a y.
    \n Notice that for integer exponents, there is the more efficient
    <code>float __builtin_powif(float x, int y)</code>. */
__ATTR_CONST__ extern float powf (float x, float y);
/** The function pow() returns the value of \a x to the exponent \a y.
    \n Notice that for integer exponents, there is the more efficient
    <code>double __builtin_powi(double x, int y)</code>. */
__ATTR_CONST__ extern double pow (double x, double y);
/** The function powl() returns the value of \a x to the exponent \a y.
    \n Notice that for integer exponents, there is the more efficient
    <code>long double __builtin_powil(long double x, int y)</code>. */
__ATTR_CONST__ extern long double powl (long double x, long double y);

/** The function isnanf() returns 1 if the argument \a x represents a
    "not-a-number" (NaN) object, otherwise 0. */
__ATTR_CONST__ extern int isnanf (float x);
/** The function isnan() returns 1 if the argument \a x represents a
    "not-a-number" (NaN) object, otherwise 0. */
__ATTR_CONST__ extern int isnan (double x);
/** The function isnanl() returns 1 if the argument \a x represents a
    "not-a-number" (NaN) object, otherwise 0. */
__ATTR_CONST__ extern int isnanl (long double x);

/** The function isinff() returns 1 if the argument \a x is positive
    infinity, &minus;1 if \a x is negative infinity, and 0 otherwise. */
__ATTR_CONST__ extern int isinff (float x);
/** The function isinf() returns 1 if the argument \a x is positive
    infinity, &minus;1 if \a x is negative infinity, and 0 otherwise. */
__ATTR_CONST__ extern int isinf (double x);
/** The function isinfl() returns 1 if the argument \a x is positive
    infinity, &minus;1 if \a x is negative infinity, and 0 otherwise. */
__ATTR_CONST__ extern int isinfl (long double x);

/** The isfinitef() function returns a nonzero value if \a __x is finite:
    not plus or minus infinity, and not NaN. */
__ATTR_CONST__ static __ATTR_ALWAYS_INLINE__ int isfinitef (float __x)
{
    unsigned char __exp;
    __asm__ (
        "mov    %0, %C1"     "\n\t"
        "lsl    %0"          "\n\t"
        "mov    %0, %D1"     "\n\t"
        "rol    %0"
        : "=&r" (__exp)
        : "r" (__x) );
    return __exp != 0xff;
}

/** The isfinite() function returns a nonzero value if \a __x is finite:
    not plus or minus infinity, and not NaN. */
#ifdef __DOXYGEN__
static __ATTR_ALWAYS_INLINE__ int isfinite (double __x);
#elif __SIZEOF_DOUBLE__ == __SIZEOF_FLOAT__
static __ATTR_ALWAYS_INLINE__ int isfinite (double __x)
{
    return isfinitef (__x);
}
#else
int isfinite (double __x);
#endif /* double = float */

/** The isfinite() function returns a nonzero value if \a __x is finite:
    not plus or minus infinity, and not NaN. */
#ifdef __DOXYGEN__
static __ATTR_ALWAYS_INLINE__ int isfinitel (long double __x);
#elif __SIZEOF_LONG_DOUBLE__ == __SIZEOF_FLOAT__
static __ATTR_ALWAYS_INLINE__ int isfinitel (long double __x)
{
    return isfinitef (__x);
}
#else
int isfinitel (long double __x);
#endif /* long double = float */

/** The copysignf() function returns \a __x but with the sign of \a __y.
    They work even if \a __x or \a __y are NaN or zero. */
__ATTR_CONST__ static __ATTR_ALWAYS_INLINE__ float copysignf (float __x, float __y)
{
    __asm__ (
        "bst    %D2, 7"  "\n\t"
        "bld    %D0, 7"
        : "=r" (__x)
        : "0" (__x), "r" (__y));
    return __x;
}

```

```
/** The copysign() function returns \a __x but with the sign of \a __y.
    They work even if \a __x or \a __y are NaN or zero. */
__ATTR_CONST__ static __ATTR_ALWAYS_INLINE__ double copysign (double __x, double __y)
{
    __asm__ (
        "bst    %r1+%2-1, 7" "\n\t"
        "bld    %r0+%2-1, 7"
        : "+r" (__x)
        : "r" (__y), "n" (__SIZEOF_DOUBLE__));
    return __x;
}

/** The copysignl() function returns \a __x but with the sign of \a __y.
    They work even if \a __x or \a __y are NaN or zero. */
__ATTR_CONST__ static __ATTR_ALWAYS_INLINE__ long double copysignl (long double __x, long double __y)
{
    __asm__ (
        "bst    %r1+%2-1, 7" "\n\t"
        "bld    %r0+%2-1, 7"
        : "+r" (__x)
        : "r" (__y), "n" (__SIZEOF_LONG_DOUBLE__));
    return __x;
}

/** The signbitf() function returns a nonzero value if the value of \a x
    has its sign bit set.  This is not the same as `\a x < 0.0',
    because IEEE 754 floating point allows zero to be signed. The
    comparison '&minus;0.0 < 0.0' is false, but `signbit (&minus;0.0)' will return a
    nonzero value. */
__ATTR_CONST__ extern int signbitf (float x);
/** The signbit() function returns a nonzero value if the value of \a x
    has its sign bit set.  This is not the same as `\a x < 0.0',
    because IEEE 754 floating point allows zero to be signed. The
    comparison '&minus;0.0 < 0.0' is false, but `signbit (&minus;0.0)' will return a
    nonzero value. */
__ATTR_CONST__ extern int signbit (double x);
/** The signbitl() function returns a nonzero value if the value of \a x
    has its sign bit set.  This is not the same as `\a x < 0.0',
    because IEEE 754 floating point allows zero to be signed. The
    comparison '&minus;0.0 < 0.0' is false, but `signbit (&minus;0.0)' will return a
    nonzero value. */
__ATTR_CONST__ extern int signbitl (long double x);

/** The fdimf() function returns <em>max(x &minus; y, 0)</em>. If \a x or
    \a y or both are NaN, NaN is returned. */
__ATTR_CONST__ extern float fdimf (float x, float y);
/** The fdim() function returns <em>max(x &minus; y, 0)</em>. If \a x or
    \a y or both are NaN, NaN is returned. */
__ATTR_CONST__ extern double fdim (double x, double y);
/** The fdiml() function returns <em>max(x &minus; y, 0)</em>. If \a x or
    \a y or both are NaN, NaN is returned. */
__ATTR_CONST__ extern long double fdiml (long double x, long double y);

/** The fmaf() function performs floating-point multiply-add. This is the
    operation <em>(x * y) + z</em>, but the intermediate result is
    not rounded to the destination type.  This can sometimes improve the
    precision of a calculation. */
__ATTR_CONST__ extern float fmaf (float x, float y, float z);
/** The fma() function performs floating-point multiply-add. This is the
    operation <em>(x * y) + z</em>, but the intermediate result is
    not rounded to the destination type.  This can sometimes improve the
    precision of a calculation. */
__ATTR_CONST__ extern double fma (double x, double y, double z);
/** The fmal() function performs floating-point multiply-add. This is the
    operation <em>(x * y) + z</em>, but the intermediate result is
    not rounded to the destination type.  This can sometimes improve the
    precision of a calculation. */
__ATTR_CONST__ extern long double fmal (long double x, long double y, long double z);

/** The fmaxf() function returns the greater of the two values \a x and
    \a y. If an argument is NaN, the other argument is returned. If
    both arguments are NaN, NaN is returned. */
__ATTR_CONST__ extern float fmaxf (float x, float y);
/** The fmax() function returns the greater of the two values \a x and
    \a y. If an argument is NaN, the other argument is returned. If
    both arguments are NaN, NaN is returned. */
__ATTR_CONST__ extern double fmax (double x, double y);
/** The fmaxl() function returns the greater of the two values \a x and
    \a y. If an argument is NaN, the other argument is returned. If
    both arguments are NaN, NaN is returned. */
__ATTR_CONST__ extern long double fmaxl (long double x, long double y);

/** The fminf() function returns the lesser of the two values \a x and
    \a y. If an argument is NaN, the other argument is returned. If
    both arguments are NaN, NaN is returned. */
__ATTR_CONST__ extern float fminf (float x, float y);
/** The fmin() function returns the lesser of the two values \a x and
```

```
    \a y. If an argument is NaN, the other argument is returned. If
    both arguments are NaN, NaN is returned. */
__ATTR_CONST__ extern double fmin (double x, double y);
/** The fminl() function returns the lesser of the two values \a x and
    \a y. If an argument is NaN, the other argument is returned. If
    both arguments are NaN, NaN is returned. */
__ATTR_CONST__ extern long double fminl (long double x, long double y);

/** The truncf() function rounds \a x to the nearest integer not larger
    in absolute value. */
__ATTR_CONST__ extern float truncf (float x);
/** The trunc() function rounds \a x to the nearest integer not larger
    in absolute value. */
__ATTR_CONST__ extern double trunc (double x);
/** The truncl() function rounds \a x to the nearest integer not larger
    in absolute value. */
__ATTR_CONST__ extern long double truncl (long double x);

/** The roundf() function rounds \a x to the nearest integer, but rounds
    halfway cases away from zero (instead of to the nearest even integer).
    Overflow is impossible.

    \return The rounded value. If \a x is an integral or infinite, \a
    x itself is returned. If \a x is \c NaN, then \c NaN is returned. */
__ATTR_CONST__ extern float roundf (float x);
/** The round() function rounds \a x to the nearest integer, but rounds
    halfway cases away from zero (instead of to the nearest even integer).
    Overflow is impossible.

    \return The rounded value. If \a x is an integral or infinite, \a
    x itself is returned. If \a x is \c NaN, then \c NaN is returned. */
__ATTR_CONST__ extern double round (double x);
/** The roundl() function rounds \a x to the nearest integer, but rounds
    halfway cases away from zero (instead of to the nearest even integer).
    Overflow is impossible.

    \return The rounded value. If \a x is an integral or infinite, \a
    x itself is returned. If \a x is \c NaN, then \c NaN is returned. */
__ATTR_CONST__ extern long double roundl (long double x);

/** The lroundf() function rounds \a x to the nearest integer, but rounds
    halfway cases away from zero (instead of to the nearest even integer).
    This function is similar to round() function, but it differs in type of
    return value and in that an overflow is possible.

    \return The rounded long integer value. If \a x is not a finite number
    or an overflow was, this realization returns the \c LONG_MIN value
    (0x80000000). */
__ATTR_CONST__ extern long lroundf (float x);
/** The lround() function rounds \a x to the nearest integer, but rounds
    halfway cases away from zero (instead of to the nearest even integer).
    This function is similar to round() function, but it differs in type of
    return value and in that an overflow is possible.

    \return The rounded long integer value. If \a x is not a finite number
    or an overflow was, this realization returns the \c LONG_MIN value
    (0x80000000). */
__ATTR_CONST__ extern long lround (double x);
/** The lroundl() function rounds \a x to the nearest integer, but rounds
    halfway cases away from zero (instead of to the nearest even integer).
    This function is similar to round() function, but it differs in type of
    return value and in that an overflow is possible.

    \return The rounded long integer value. If \a x is not a finite number
    or an overflow was, this realization returns the \c LONG_MIN value
    (0x80000000). */
__ATTR_CONST__ extern long lroundl (long double x);

/** The lrintf() function rounds \a x to the nearest integer, rounding the
    halfway cases to the even integer direction. (That is both 1.5 and 2.5
    values are rounded to 2). This function is similar to rintf() function,
    but it differs in type of return value and in that an overflow is
    possible.

    \return The rounded long integer value. If \a x is not a finite
    number or an overflow was, this realization returns the \c LONG_MIN
    value (0x80000000). */
__ATTR_CONST__ extern long lrintf (float x);
/** The lrint() function rounds \a x to the nearest integer, rounding the
    halfway cases to the even integer direction. (That is both 1.5 and 2.5
    values are rounded to 2). This function is similar to rint() function,
    but it differs in type of return value and in that an overflow is
    possible.

    \return The rounded long integer value. If \a x is not a finite
    number or an overflow was, this realization returns the \c LONG_MIN
    value (0x80000000). */
```

```
__ATTR_CONST__ extern long lrint (double x);
/** The lrintl() function rounds \a x to the nearest integer, rounding the
    halfway cases to the even integer direction. (That is both 1.5 and 2.5
    values are rounded to 2). This function is similar to rintl() function,
    but it differs in type of return value and in that an overflow is
    possible.

    \return The rounded long integer value. If \a x is not a finite
    number or an overflow was, this realization returns the \c LONG_MIN
    value (0x80000000). */
__ATTR_CONST__ extern long lrintl (long double x);

/**@}*/

/**@{*/
/**
   \name Non-Standard Math Functions
*/

/** \ingroup avr_math
    The function squaref() returns <em>x * x</em>.
    \note This function does not belong to the C standard definition. */
__ATTR_CONST__ extern float squaref (float x);

/** The function square() returns <em>x * x</em>.
    \note This function does not belong to the C standard definition. */

#if defined(__DOXYGEN__) || __SIZEOF_DOUBLE__ == __SIZEOF_FLOAT__
__ATTR_CONST__ extern double square (double x);
#elif defined(__WITH_LIBF7_MATH__)
__ATTR_CONST__ extern double square (double x) __asm("__f7_square");
#endif

/** The function squarel() returns <em>x * x</em>.
    \note This function does not belong to the C standard definition. */
#if defined(__DOXYGEN__) || __SIZEOF_LONG_DOUBLE__ == __SIZEOF_FLOAT__
__ATTR_CONST__ extern long double squarel (long double x);
#elif defined(__WITH_LIBF7_MATH__)
__ATTR_CONST__ extern long double squarel (long double x) __asm("__f7_square");
#endif

/**@}*/

#ifdef __cplusplus
}
#endif

#endif /* !__MATH_H */
```

23.50 setjmp.h File Reference

Functions

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

23.51 setjmp.h

Go to the documentation of this file.

```
/* Copyright (c) 2002,2007 Marek Michalkiewicz
   All rights reserved.

   Redistribution and use in source and binary forms, with or without
   modification, are permitted provided that the following conditions are met:

   * Redistributions of source code must retain the above copyright
     notice, this list of conditions and the following disclaimer.

   * Redistributions in binary form must reproduce the above copyright
     notice, this list of conditions and the following disclaimer in
     the documentation and/or other materials provided with the
     distribution.

   * Neither the name of the copyright holders nor the names of
     contributors may be used to endorse or promote products derived
     from this software without specific prior written permission.
```

```
   THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
   AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
   IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
   ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
   LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
   CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
   SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
   INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
   CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
   ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
   POSSIBILITY OF SUCH DAMAGE. */

/* $Id$ */

#ifndef __SETJMP_H_
#define __SETJMP_H_ 1

#ifdef __cplusplus
extern "C" {
#endif

/*
   jmp_buf:
    offset  size    description
     0  16/2    call-saved registers (r2-r17)
                (AVR_TINY arch has only 2 call saved registers (r18,r19))
    16/2     2  frame pointer (r29:r28)
    18/4     2  stack pointer (SPH:SPL)
    20/6     1  status register (SREG)
    21/7     2/3    return address (PC) (2 bytes used for <=128Kw flash)
    23/24/9 = total size (AVR_TINY arch always has 2 bytes PC)
 */

#if !defined(__DOXYGEN__)

#if defined(__AVR_TINY__)
# define _JBLEN  9
#elif   defined(__AVR_3_BYTE_PC__) && __AVR_3_BYTE_PC__
# define _JBLEN  24
#else
# define _JBLEN  23
#endif
typedef struct _jmp_buf { unsigned char _jb[_JBLEN]; } jmp_buf[1];

#endif /* not __DOXYGEN__ */

/** \file */
/** \defgroup setjmp <setjmp.h>: Non-local goto

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

    \note #setjmp and #longjmp make programs hard to understand and maintain.
    If possible, an alternative should be used.

    \note longjmp can destroy changes made to global register
    variables (see \ref faq_regbind).

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

    Example:

    \code
    #include <setjmp.h>

    jmp_buf env;

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

        while (1)
        {
           // Main processing loop which calls foo() somewhere ...
        }
    }

```

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

        if (err)
        {
            longjmp (env, 1);
        }
    }
    \endcode */

#ifndef __DOXYGEN__
#ifndef __ATTR_NORETURN__
#define __ATTR_NORETURN__ __attribute__((__noreturn__))
#endif
#endif /* ! DOXYGEN */

/** \ingroup setjmp
    \brief Save stack context for non-local goto.

    \code #include <setjmp.h>\endcode

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

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

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

extern int setjmp(jmp_buf __jmpb);

/** \ingroup setjmp
    \brief Non-local jump to a saved stack context.

    \code #include <setjmp.h>\endcode

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

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

    \param __jmpb Information saved by a previous call to setjmp().
    \param __ret  Value to return to the caller of setjmp().

    \returns This function never returns. */

extern void longjmp(jmp_buf __jmpb, int __ret) __ATTR_NORETURN__;

#ifdef __cplusplus
}
#endif

#endif  /* !__SETJMP_H_ */
```

23.52 stdint.h File Reference

Macros

Limits of specified-width integer types

C++ implementations should define these macros only when __STDC_LIMIT_MACROS is defined before <stdint.h> is included

- #define INT8_MAX 0x7f
- #define INT8_MIN (-INT8_MAX - 1)
- #define UINT8_MAX (INT8_MAX * 2 + 1)
- #define INT16_MAX 0x7fff
- #define INT16_MIN (-INT16_MAX - 1)
- #define UINT16_MAX (__CONCAT(INT16_MAX, U) * 2U + 1U)
- #define INT32_MAX 0x7fffffffL
- #define INT32_MIN (-INT32_MAX - 1L)
- #define UINT32_MAX (__CONCAT(INT32_MAX, U) * 2UL + 1UL)

- #define INT64_MAX 0x7fffffffffffffffLL
- #define INT64_MIN (-INT64_MAX - 1LL)
- #define UINT64_MAX (__CONCAT(INT64_MAX, U) ∗ 2ULL + 1ULL)

Limits of minimum-width integer types

- #define INT_LEAST8_MAX INT8_MAX
- #define INT_LEAST8_MIN INT8_MIN
- #define UINT_LEAST8_MAX UINT8_MAX
- #define INT_LEAST16_MAX INT16_MAX
- #define INT_LEAST16_MIN INT16_MIN
- #define UINT_LEAST16_MAX UINT16_MAX
- #define INT_LEAST32_MAX INT32_MAX
- #define INT_LEAST32_MIN INT32_MIN
- #define UINT_LEAST32_MAX UINT32_MAX
- #define INT_LEAST64_MAX INT64_MAX
- #define INT_LEAST64_MIN INT64_MIN
- #define UINT_LEAST64_MAX UINT64_MAX

Limits of fastest minimum-width integer types

- #define INT_FAST8_MAX INT8_MAX
- #define INT_FAST8_MIN INT8_MIN
- #define UINT_FAST8_MAX UINT8_MAX
- #define INT_FAST16_MAX INT16_MAX
- #define INT_FAST16_MIN INT16_MIN
- #define UINT_FAST16_MAX UINT16_MAX
- #define INT_FAST32_MAX INT32_MAX
- #define INT_FAST32_MIN INT32_MIN
- #define UINT_FAST32_MAX UINT32_MAX
- #define INT_FAST64_MAX INT64_MAX
- #define INT_FAST64_MIN INT64_MIN
- #define UINT_FAST64_MAX UINT64_MAX

Limits of integer types capable of holding object pointers

- #define INTPTR_MAX INT16_MAX
- #define INTPTR_MIN INT16_MIN
- #define UINTPTR_MAX UINT16_MAX

Limits of greatest-width integer types

- #define INTMAX_MAX INT64_MAX
- #define INTMAX_MIN INT64_MIN
- #define UINTMAX_MAX UINT64_MAX

Limits of other integer types

C++ implementations should define these macros only when __STDC_LIMIT_MACROS is defined before <stdint.h> is included

- #define PTRDIFF_MAX INT16_MAX
- #define PTRDIFF_MIN INT16_MIN
- #define SIG_ATOMIC_MAX INT8_MAX
- #define SIG_ATOMIC_MIN INT8_MIN
- #define SIZE_MAX UINT16_MAX
- #define **WCHAR_MAX** __WCHAR_MAX__
- #define **WCHAR_MIN** __WCHAR_MIN__
- #define **WINT_MAX** __WINT_MAX__
- #define **WINT_MIN** __WINT_MIN__

Macros for integer constants

C++ implementations should define these macros only when __STDC_CONSTANT_MACROS is defined before <stdint.h> is included.

These definitions are valid for integer constants without suffix and for macros defined as integer constant without suffix

- #define INT8_C(value) ((int8_t) value)
- #define UINT8_C(value) ((uint8_t) __CONCAT(value, U))
- #define INT16_C(value) value
- #define UINT16_C(value) __CONCAT(value, U)
- #define INT32_C(value) __CONCAT(value, L)
- #define UINT32_C(value) __CONCAT(value, UL)
- #define INT64_C(value) __CONCAT(value, LL)
- #define UINT64_C(value) __CONCAT(value, ULL)
- #define INTMAX_C(value) __CONCAT(value, LL)
- #define UINTMAX_C(value) __CONCAT(value, ULL)

Typedefs

Exact-width integer types

Integer types having exactly the specified width

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

Integer types capable of holding object pointers

These allow you to declare variables of the same size as a pointer.

- typedef int16_t intptr_t
- typedef uint16_t uintptr_t

Minimum-width integer types

Integer types having at least the specified width

- typedef int8_t int_least8_t
- typedef uint8_t uint_least8_t
- typedef int16_t int_least16_t
- typedef uint16_t uint_least16_t
- typedef int32_t int_least32_t
- typedef uint32_t uint_least32_t
- typedef int64_t int_least64_t
- typedef uint64_t uint_least64_t

Fastest minimum-width integer types

Integer types being usually fastest having at least the specified width

- typedef int8_t int_fast8_t
- typedef uint8_t uint_fast8_t
- typedef int16_t int_fast16_t
- typedef uint16_t uint_fast16_t
- typedef int32_t int_fast32_t
- typedef uint32_t uint_fast32_t

- typedef int64_t int_fast64_t
- typedef uint64_t uint_fast64_t

Greatest-width integer types

Types designating integer data capable of representing any value of any integer type in the corresponding signed or unsigned category

- typedef int64_t intmax_t
- typedef uint64_t uintmax_t

23.53 stdint.h

Go to the documentation of this file.

```
/* Copyright (c) 2002,2004,2005 Marek Michalkiewicz
   Copyright (c) 2005, Carlos Lamas
   Copyright (c) 2005,2007 Joerg Wunsch
   Copyright (c) 2013 Embecosm
   All rights reserved.

   Redistribution and use in source and binary forms, with or without
   modification, are permitted provided that the following conditions are met:

   * Redistributions of source code must retain the above copyright
     notice, this list of conditions and the following disclaimer.

   * Redistributions in binary form must reproduce the above copyright
     notice, this list of conditions and the following disclaimer in
     the documentation and/or other materials provided with the
     distribution.

   * Neither the name of the copyright holders nor the names of
     contributors may be used to endorse or promote products derived
     from this software without specific prior written permission.

  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
  AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
  IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
  ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
  LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
  CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
  SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
  INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
  CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
  ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
  POSSIBILITY OF SUCH DAMAGE. */

/* $Id$ */

/*
 * ISO/IEC 9899:1999  7.18 Integer types <stdint.h>
 */

#ifndef __STDINT_H_
#define __STDINT_H_

/** \file */
/** \defgroup avr_stdint <stdint.h>: Standard Integer Types
    \code #include <stdint.h> \endcode

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

    Since these typedefs are mandated by the C99 standard, they are preferred
    over rolling your own typedefs.  */

#ifndef __DOXYGEN__
/*
 * __USING_MINT8 is defined to 1 if the -mint8 option is in effect.
 */
#if __INT_MAX__ == 127
# define __USING_MINT8 1
#else
# define __USING_MINT8 0
#endif

#endif  /* !__DOXYGEN__ */

/* Integer types */
```

```

#if defined(__DOXYGEN__)

/* doxygen gets confused by the __attribute__ stuff */

/** \name Exact-width integer types
    Integer types having exactly the specified width */

/**@{*/

/** \ingroup avr_stdint
    8-bit signed type. */

typedef signed char int8_t;

/** \ingroup avr_stdint
    8-bit unsigned type. */

typedef unsigned char uint8_t;

/** \ingroup avr_stdint
    16-bit signed type. */

typedef signed int int16_t;

/** \ingroup avr_stdint
    16-bit unsigned type. */

typedef unsigned int uint16_t;

/** \ingroup avr_stdint
    32-bit signed type. */

typedef signed long int int32_t;

/** \ingroup avr_stdint
    32-bit unsigned type. */

typedef unsigned long int uint32_t;

/** \ingroup avr_stdint
    64-bit signed type.
    \note This type is not available when the compiler
    option -mint8 is in effect. */

typedef signed long long int int64_t;

/** \ingroup avr_stdint
    64-bit unsigned type.
    \note This type is not available when the compiler
    option -mint8 is in effect. */

typedef unsigned long long int uint64_t;

/**@}*/

#else /* !defined(__DOXYGEN__) */

/* actual implementation goes here */

typedef signed int int8_t __attribute__((__mode__(__QI__)));
typedef unsigned int uint8_t __attribute__((__mode__(__QI__)));
typedef signed int int16_t __attribute__ ((__mode__ (__HI__)));
typedef unsigned int uint16_t __attribute__ ((__mode__ (__HI__)));
typedef signed int int32_t __attribute__ ((__mode__ (__SI__)));
typedef unsigned int uint32_t __attribute__ ((__mode__ (__SI__)));
#if !__USING_MINT8
typedef signed int int64_t __attribute__((__mode__(__DI__)));
typedef unsigned int uint64_t __attribute__((__mode__(__DI__)));
#endif

#endif /* defined(__DOXYGEN__) */

/** \name Integer types capable of holding object pointers
    These allow you to declare variables of the same size as a pointer. */

/**@{*/

/** \ingroup avr_stdint
    Signed pointer compatible type. */

typedef int16_t intptr_t;

/** \ingroup avr_stdint
    Unsigned pointer compatible type. */

typedef uint16_t uintptr_t;
```

```

/**@}*/

/** \name Minimum-width integer types
   Integer types having at least the specified width */

/**@{*/

/** \ingroup avr_stdint
    signed int with at least 8 bits. */

typedef int8_t   int_least8_t;

/** \ingroup avr_stdint
    unsigned int with at least 8 bits. */

typedef uint8_t  uint_least8_t;

/** \ingroup avr_stdint
    signed int with at least 16 bits. */

typedef int16_t  int_least16_t;

/** \ingroup avr_stdint
    unsigned int with at least 16 bits. */

typedef uint16_t uint_least16_t;

/** \ingroup avr_stdint
    signed int with at least 32 bits. */

typedef int32_t  int_least32_t;

/** \ingroup avr_stdint
    unsigned int with at least 32 bits. */

typedef uint32_t uint_least32_t;

#if !__USING_MINT8 || defined(__DOXYGEN__)
/** \ingroup avr_stdint
    signed int with at least 64 bits.
    \note This type is not available when the compiler
    option -mint8 is in effect. */

typedef int64_t  int_least64_t;

/** \ingroup avr_stdint
    unsigned int with at least 64 bits.
    \note This type is not available when the compiler
    option -mint8 is in effect. */

typedef uint64_t uint_least64_t;
#endif

/**@}*/


/** \name Fastest minimum-width integer types
   Integer types being usually fastest having at least the specified width */

/**@{*/

/** \ingroup avr_stdint
    fastest signed int with at least 8 bits. */

typedef int8_t int_fast8_t;

/** \ingroup avr_stdint
    fastest unsigned int with at least 8 bits. */

typedef uint8_t uint_fast8_t;

/** \ingroup avr_stdint
    fastest signed int with at least 16 bits. */

typedef int16_t int_fast16_t;

/** \ingroup avr_stdint
    fastest unsigned int with at least 16 bits. */

typedef uint16_t uint_fast16_t;

/** \ingroup avr_stdint
    fastest signed int with at least 32 bits. */

typedef int32_t int_fast32_t;

```

```
/** \ingroup avr_stdint
    fastest unsigned int with at least 32 bits. */

typedef uint32_t uint_fast32_t;

#if !__USING_MINT8 || defined(__DOXYGEN__)
/** \ingroup avr_stdint
    fastest signed int with at least 64 bits.
    \note This type is not available when the compiler
    option -mint8 is in effect. */

typedef int64_t int_fast64_t;

/** \ingroup avr_stdint
    fastest unsigned int with at least 64 bits.
    \note This type is not available when the compiler
    option -mint8 is in effect. */

typedef uint64_t uint_fast64_t;
#endif

/**@}*/


/** \name Greatest-width integer types
   Types designating integer data capable of representing any value of
   any integer type in the corresponding signed or unsigned category */

/**@{*/

#if __USING_MINT8
typedef int32_t intmax_t;

typedef uint32_t uintmax_t;
#else  /* !__USING_MINT8 */
/** \ingroup avr_stdint
    largest signed int available. */

typedef int64_t intmax_t;

/** \ingroup avr_stdint
    largest unsigned int available. */

typedef uint64_t uintmax_t;
#endif /* __USING_MINT8 */

/**@}*/

#ifndef __DOXYGEN__
/* Helping macro */
#ifndef __CONCAT
#define __CONCATenate(left, right) left ## right
#define __CONCAT(left, right) __CONCATenate(left, right)
#endif

#endif  /* !__DOXYGEN__ */

#if !defined(__cplusplus) || defined(__STDC_LIMIT_MACROS)

/** \name Limits of specified-width integer types
   C++ implementations should define these macros only when
   __STDC_LIMIT_MACROS is defined before <stdint.h> is included */

/**@{*/

/** \ingroup avr_stdint
    largest positive value an int8_t can hold. */

#define INT8_MAX 0x7f

/** \ingroup avr_stdint
    smallest negative value an int8_t can hold. */

#define INT8_MIN (-INT8_MAX - 1)

#if __USING_MINT8

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

#define INT16_MAX 0x7fffL
#define INT16_MIN (-INT16_MAX - 1L)
#define UINT16_MAX (__CONCAT(INT16_MAX, U) * 2UL + 1UL)

#define INT32_MAX 0x7fffffffLL
#define INT32_MIN (-INT32_MAX - 1LL)
#define UINT32_MAX (__CONCAT(INT32_MAX, U) * 2ULL + 1ULL)

```

```
#else /* !__USING_MINT8 */

/** \ingroup avr_stdint
    largest value an uint8_t can hold. */

#define UINT8_MAX (INT8_MAX * 2 + 1)

/** \ingroup avr_stdint
    largest positive value an int16_t can hold. */

#define INT16_MAX 0x7fff

/** \ingroup avr_stdint
    smallest negative value an int16_t can hold. */

#define INT16_MIN (-INT16_MAX - 1)

/** \ingroup avr_stdint
    largest value an uint16_t can hold. */

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

/** \ingroup avr_stdint
    largest positive value an int32_t can hold. */

#define INT32_MAX 0x7fffffffL

/** \ingroup avr_stdint
    smallest negative value an int32_t can hold. */

#define INT32_MIN (-INT32_MAX - 1L)

/** \ingroup avr_stdint
    largest value an uint32_t can hold. */

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

#endif /* __USING_MINT8 */

/** \ingroup avr_stdint
    largest positive value an int64_t can hold. */

#define INT64_MAX 0x7fffffffffffffffLL

/** \ingroup avr_stdint
    smallest negative value an int64_t can hold. */

#define INT64_MIN (-INT64_MAX - 1LL)

/** \ingroup avr_stdint
    largest value an uint64_t can hold. */

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

/**@}*/

/** \name Limits of minimum-width integer types */
/**@{*/

/** \ingroup avr_stdint
    largest positive value an int_least8_t can hold. */

#define INT_LEAST8_MAX INT8_MAX

/** \ingroup avr_stdint
    smallest negative value an int_least8_t can hold. */

#define INT_LEAST8_MIN INT8_MIN

/** \ingroup avr_stdint
    largest value an uint_least8_t can hold. */

#define UINT_LEAST8_MAX UINT8_MAX

/** \ingroup avr_stdint
    largest positive value an int_least16_t can hold. */

#define INT_LEAST16_MAX INT16_MAX

/** \ingroup avr_stdint
    smallest negative value an int_least16_t can hold. */

#define INT_LEAST16_MIN INT16_MIN

/** \ingroup avr_stdint
    largest value an uint_least16_t can hold. */

```

```
#define UINT_LEAST16_MAX UINT16_MAX

/** \ingroup avr_stdint
    largest positive value an int_least32_t can hold. */

#define INT_LEAST32_MAX INT32_MAX

/** \ingroup avr_stdint
    smallest negative value an int_least32_t can hold. */

#define INT_LEAST32_MIN INT32_MIN

/** \ingroup avr_stdint
    largest value an uint_least32_t can hold. */

#define UINT_LEAST32_MAX UINT32_MAX

/** \ingroup avr_stdint
    largest positive value an int_least64_t can hold. */

#define INT_LEAST64_MAX INT64_MAX

/** \ingroup avr_stdint
    smallest negative value an int_least64_t can hold. */

#define INT_LEAST64_MIN INT64_MIN

/** \ingroup avr_stdint
    largest value an uint_least64_t can hold. */

#define UINT_LEAST64_MAX UINT64_MAX

/**@}*/

/** \name Limits of fastest minimum-width integer types */

/**@{*/

/** \ingroup avr_stdint
    largest positive value an int_fast8_t can hold. */

#define INT_FAST8_MAX INT8_MAX

/** \ingroup avr_stdint
    smallest negative value an int_fast8_t can hold. */

#define INT_FAST8_MIN INT8_MIN

/** \ingroup avr_stdint
    largest value an uint_fast8_t can hold. */

#define UINT_FAST8_MAX UINT8_MAX

/** \ingroup avr_stdint
    largest positive value an int_fast16_t can hold. */

#define INT_FAST16_MAX INT16_MAX

/** \ingroup avr_stdint
    smallest negative value an int_fast16_t can hold. */

#define INT_FAST16_MIN INT16_MIN

/** \ingroup avr_stdint
    largest value an uint_fast16_t can hold. */

#define UINT_FAST16_MAX UINT16_MAX

/** \ingroup avr_stdint
    largest positive value an int_fast32_t can hold. */

#define INT_FAST32_MAX INT32_MAX

/** \ingroup avr_stdint
    smallest negative value an int_fast32_t can hold. */

#define INT_FAST32_MIN INT32_MIN

/** \ingroup avr_stdint
    largest value an uint_fast32_t can hold. */

#define UINT_FAST32_MAX UINT32_MAX

/** \ingroup avr_stdint
    largest positive value an int_fast64_t can hold. */

#define INT_FAST64_MAX INT64_MAX
```

```

/** \ingroup avr_stdint
    smallest negative value an int_fast64_t can hold. */

#define INT_FAST64_MIN INT64_MIN

/** \ingroup avr_stdint
    largest value an uint_fast64_t can hold. */

#define UINT_FAST64_MAX UINT64_MAX

/**@}*/

/** \name Limits of integer types capable of holding object pointers */

/**@{*/

/** \ingroup avr_stdint
    largest positive value an intptr_t can hold. */

#define INTPTR_MAX INT16_MAX

/** \ingroup avr_stdint
    smallest negative value an intptr_t can hold. */

#define INTPTR_MIN INT16_MIN

/** \ingroup avr_stdint
    largest value an uintptr_t can hold. */

#define UINTPTR_MAX UINT16_MAX

/**@}*/

/** \name Limits of greatest-width integer types */

/**@{*/

/** \ingroup avr_stdint
    largest positive value an intmax_t can hold. */

#define INTMAX_MAX INT64_MAX

/** \ingroup avr_stdint
    smallest negative value an intmax_t can hold. */

#define INTMAX_MIN INT64_MIN

/** \ingroup avr_stdint
    largest value an uintmax_t can hold. */

#define UINTMAX_MAX UINT64_MAX

/**@}*/

/** \name Limits of other integer types
    C++ implementations should define these macros only when
    __STDC_LIMIT_MACROS is defined before <stdint.h> is included */

/**@{*/

/** \ingroup avr_stdint
    largest positive value a ptrdiff_t can hold. */

#define PTRDIFF_MAX INT16_MAX

/** \ingroup avr_stdint
    smallest negative value a ptrdiff_t can hold. */

#define PTRDIFF_MIN INT16_MIN


/* Limits of sig_atomic_t */
/* signal.h is currently not implemented (not avr/signal.h) */

/** \ingroup avr_stdint
    largest positive value a sig_atomic_t can hold. */

#define SIG_ATOMIC_MAX INT8_MAX

/** \ingroup avr_stdint
    smallest negative value a sig_atomic_t can hold. */

#define SIG_ATOMIC_MIN INT8_MIN


/** \ingroup avr_stdint
```

```
    largest value a size_t can hold. */

#define SIZE_MAX UINT16_MAX


/* Limits of wchar_t */
/* wchar.h is currently not implemented */
/* #define WCHAR_MAX */
/* #define WCHAR_MIN */


/* Limits of wint_t */
/* wchar.h is currently not implemented */
#ifndef WCHAR_MAX
#define WCHAR_MAX __WCHAR_MAX__
#define WCHAR_MIN __WCHAR_MIN__
#endif
#ifndef WINT_MAX
#define WINT_MAX __WINT_MAX__
#define WINT_MIN __WINT_MIN__
#endif


#endif /* !defined(__cplusplus) || defined(__STDC_LIMIT_MACROS) */

#if (!defined __cplusplus || __cplusplus >= 201103L \
     || defined __STDC_CONSTANT_MACROS)

/** \name Macros for integer constants
    C++ implementations should define these macros only when
    __STDC_CONSTANT_MACROS is defined before <stdint.h> is included.

    These definitions are valid for integer constants without suffix and
    for macros defined as integer constant without suffix */

/* The GNU C preprocessor defines special macros in the implementation
   namespace to allow a definition that works in #if expressions.  */
#ifdef __INT8_C
#define INT8_C(c) __INT8_C(c)
#define INT16_C(c) __INT16_C(c)
#define INT32_C(c) __INT32_C(c)
#define INT64_C(c) __INT64_C(c)
#define UINT8_C(c) __UINT8_C(c)
#define UINT16_C(c) __UINT16_C(c)
#define UINT32_C(c) __UINT32_C(c)
#define UINT64_C(c) __UINT64_C(c)
#define INTMAX_C(c) __INTMAX_C(c)
#define UINTMAX_C(c) __UINTMAX_C(c)
#else
/** \ingroup avr_stdint
    define a constant of type int8_t */

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

/** \ingroup avr_stdint
    define a constant of type uint8_t */

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

#if __USING_MINT8

#define INT16_C(value) __CONCAT(value, L)
#define UINT16_C(value) __CONCAT(value, UL)

#define INT32_C(value) ((int32_t) __CONCAT(value, LL))
#define UINT32_C(value) ((uint32_t) __CONCAT(value, ULL))

#else /* !__USING_MINT8 */

/** \ingroup avr_stdint
    define a constant of type int16_t */

#define INT16_C(value) value

/** \ingroup avr_stdint
    define a constant of type uint16_t */

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

/** \ingroup avr_stdint
    define a constant of type int32_t */

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

/** \ingroup avr_stdint
    define a constant of type uint32_t */

```

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

#endif /* __USING_MINT8 */

/** \ingroup avr_stdint
    define a constant of type int64_t */

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

/** \ingroup avr_stdint
    define a constant of type uint64_t */

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

/** \ingroup avr_stdint
    define a constant of type intmax_t */

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

/** \ingroup avr_stdint
    define a constant of type uintmax_t */

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

#endif /* !__INT8_C */

/**@}*/

#endif /* (!defined __cplusplus || __cplusplus >= 201103L \
       || defined __STDC_CONSTANT_MACROS) */


#endif /* _STDINT_H_ */
```

23.54 stdio.h File Reference

Macros

- #define stdin (__iob[0])
- #define stdout (__iob[1])
- #define stderr (__iob[2])
- #define EOF (-1)
- #define fdev_set_udata(stream, u) do { (stream)->udata = u; } while(0)
- #define fdev_get_udata(stream) ((stream)->udata)
- #define fdev_setup_stream(stream, put, get, rwflag)
- #define _FDEV_SETUP_READ __SRD
- #define _FDEV_SETUP_WRITE __SWR
- #define _FDEV_SETUP_RW (__SRD|__SWR)
- #define _FDEV_ERR (-1)
- #define _FDEV_EOF (-2)
- #define FDEV_SETUP_STREAM(put, get, rwflag)
- #define fdev_close()
- #define putc(__c, __stream) fputc(__c, __stream)
- #define putchar(__c) fputc(__c, stdout)
- #define getc(__stream) fgetc(__stream)
- #define getchar() fgetc(stdin)

Typedefs

- typedef struct __file FILE

Functions

- int fclose (FILE *__stream)
- int vfprintf (FILE *__stream, const char *__fmt, va_list __ap)
- int vfprintf_P (FILE *__stream, const char *__fmt, va_list __ap)
- int fputc (int __c, FILE *__stream)
- int printf (const char *__fmt,...)
- int printf_P (const char *__fmt,...)
- int vprintf (const char *__fmt, va_list __ap)
- int sprintf (char *__s, const char *__fmt,...)
- int sprintf_P (char *__s, const char *__fmt,...)
- int snprintf (char *__s, size_t __n, const char *__fmt,...)
- int snprintf_P (char *__s, size_t __n, const char *__fmt,...)
- int vsprintf (char *__s, const char *__fmt, va_list ap)
- int vsprintf_P (char *__s, const char *__fmt, va_list ap)
- int vsnprintf (char *__s, size_t __n, const char *__fmt, va_list ap)
- int vsnprintf_P (char *__s, size_t __n, const char *__fmt, va_list ap)
- int fprintf (FILE *__stream, const char *__fmt,...)
- int fprintf_P (FILE *__stream, const char *__fmt,...)
- int fputs (const char *__str, FILE *__stream)
- int fputs_P (const char *__str, FILE *__stream)
- int puts (const char *__str)
- int puts_P (const char *__str)
- size_t fwrite (const void *__ptr, size_t __size, size_t __nmemb, FILE *__stream)
- int fgetc (FILE *__stream)
- int ungetc (int __c, FILE *__stream)
- char * fgets (char *__str, int __size, FILE *__stream)
- char * gets (char *__str)
- size_t fread (void *__ptr, size_t __size, size_t __nmemb, FILE *__stream)
- void clearerr (FILE *__stream)
- int feof (FILE *__stream)
- int ferror (FILE *__stream)
- int vfscanf (FILE *__stream, const char *__fmt, va_list __ap)
- int vfscanf_P (FILE *__stream, const char *__fmt, va_list __ap)
- int fscanf (FILE *__stream, const char *__fmt,...)
- int fscanf_P (FILE *__stream, const char *__fmt,...)
- int scanf (const char *__fmt,...)
- int scanf_P (const char *__fmt,...)
- int vscanf (const char *__fmt, va_list __ap)
- int sscanf (const char *__buf, const char *__fmt,...)
- int sscanf_P (const char *__buf, const char *__fmt,...)
- int fflush (FILE *stream)

23.55 stdio.h

Go to the documentation of this file.

```
/* Copyright (c) 2002, 2005, 2007 Joerg Wunsch
   All rights reserved.

   Portions of documentation Copyright (c) 1990, 1991, 1993
   The Regents of the University of California.

   All rights reserved.

   Redistribution and use in source and binary forms, with or without
   modification, are permitted provided that the following conditions are met:

   * Redistributions of source code must retain the above copyright
```

```
     notice, this list of conditions and the following disclaimer.

   * Redistributions in binary form must reproduce the above copyright
     notice, this list of conditions and the following disclaimer in
     the documentation and/or other materials provided with the
     distribution.

   * Neither the name of the copyright holders nor the names of
     contributors may be used to endorse or promote products derived
     from this software without specific prior written permission.

  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
  AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
  IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
  ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
  LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
  CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
  SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
  INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
  CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
  ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
  POSSIBILITY OF SUCH DAMAGE.

  $Id$
*/

#ifndef _STDIO_H_
#define _STDIO_H_ 1

#ifndef __ASSEMBLER__

#include <inttypes.h>
#include <stdarg.h>

#ifndef __DOXYGEN__
#define __need_NULL
#define __need_size_t
#include <stddef.h>
#endif  /* !__DOXYGEN__ */

/** \file */
/** \defgroup avr_stdio <stdio.h>: Standard IO facilities
    \code #include <stdio.h> \endcode

    <h3>Introduction to the Standard IO facilities</h3>

    This file declares the standard IO facilities that are implemented
    in AVR-LibC.  Due to the nature of the underlying hardware,
    only a limited subset of standard IO is implemented.  There is no
    actual file implementation available, so only device IO can be
    performed.  Since there's no operating system, the application
    needs to provide enough details about their devices in order to
    make them usable by the standard IO facilities.

    Due to space constraints, some functionality has not been
    implemented at all (like some of the \c printf conversions that
    have been left out).  Nevertheless, potential users of this
    implementation should be warned: the \c printf and \c scanf families of functions, although
    usually associated with presumably simple things like the
    famous "Hello, world!" program, are actually fairly complex
    which causes their inclusion to eat up a fair amount of code space.
    Also, they are not fast due to the nature of interpreting the
    format string at run-time.  Whenever possible, resorting to the
    (sometimes non-standard) predetermined conversion facilities that are
    offered by AVR-LibC will usually cost much less in terms of speed
    and code size.

    <h3>Tunable options for code size vs. feature set</h3>

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

    <h3>Outline of the chosen API</h3>

    The standard streams \c stdin, \c stdout, and \c stderr are
    provided, but contrary to the C standard, since AVR-LibC has no
    knowledge about applicable devices, these streams are not already
    pre-initialized at application startup.  Also, since there is no
    notion of "file" whatsoever to AVR-LibC, there is no function
    \c fopen() that could be used to associate a stream to some device.
    (See \ref stdio_note1 "note 1".)  Instead, the function \c fdevopen()
    is provided to associate a stream to a device, where the device
    needs to provide a function to send a character, to receive a
    character, or both.  There is no differentiation between "text" and
```

```
"binary" streams inside AVR-LibC.  Character \c \\n is sent
literally down to the device's \c put() function.  If the device
requires a carriage return (\c \\r) character to be sent before
the linefeed, its \c put() routine must implement this (see
\ref stdio_note2 "note 2").

As an alternative method to fdevopen(), the macro
fdev_setup_stream() might be used to setup a user-supplied FILE
structure.

It should be noted that the automatic conversion of a newline
character into a carriage return - newline sequence breaks binary
transfers.  If binary transfers are desired, no automatic
conversion should be performed, but instead any string that aims
to issue a CR-LF sequence must use <tt>"\r\n"</tt> explicitly.

For convenience, the first call to \c fdevopen() that opens a
stream for reading will cause the resulting stream to be aliased
to \c stdin.  Likewise, the first call to \c fdevopen() that opens
a stream for writing will cause the resulting stream to be aliased
to both, \c stdout, and \c stderr.  Thus, if the open was done
with both, read and write intent, all three standard streams will
be identical.  Note that these aliases are indistinguishable from
each other, thus calling \c fclose() on such a stream will also
effectively close all of its aliases (\ref stdio_note3 "note 3").

It is possible to tie additional user data to a stream, using
fdev_set_udata().  The backend put and get functions can then
extract this user data using fdev_get_udata(), and act
appropriately.  For example, a single put function could be used
to talk to two different UARTs that way, or the put and get
functions could keep internal state between calls there.

<h3>Format strings in flash ROM</h3>

All the \c printf and \c scanf family functions come in two flavours: the
standard name, where the format string is expected to be in
SRAM, as well as a version with the suffix "_P" where the format
string is expected to reside in the flash ROM.  The macro
#PSTR (explained in \ref avr_pgmspace) becomes very handy
for declaring these format strings.

\anchor stdio_without_malloc
<h3>Running stdio without malloc()</h3>

By default, fdevopen() requires malloc().  As this is often
not desired in the limited environment of a microcontroller, an
alternative option is provided to run completely without malloc().

The macro fdev_setup_stream() is provided to prepare a
user-supplied FILE buffer for operation with stdio.

<h4>Example</h4>

\code
#include <stdio.h>

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

static FILE mystdout = FDEV_SETUP_STREAM(uart_putchar, NULL,
                                         _FDEV_SETUP_WRITE);

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

  if (c == '\n')
    uart_putchar('\r', stream);
  loop_until_bit_is_set(UCSRA, UDRE);
  UDR = c;
  return 0;
}

int
main(void)
{
  init_uart();
  stdout = &mystdout;
  printf("Hello, world!\n");

  return 0;
}
\endcode

This example uses the initializer form FDEV_SETUP_STREAM() rather
than the function-like fdev_setup_stream(), so all data
initialization happens during C start-up.
```

```

    If streams initialized that way are no longer needed, they can be
    destroyed by first calling the macro fdev_close(), and then
    destroying the object itself.  No call to fclose() should be
    issued for these streams.  While calling fclose() itself is
    harmless, it will cause an undefined reference to free() and thus
    cause the linker to link the malloc module into the application.

    <h3>Notes</h3>

<dl>
<dt>\anchor stdio_note1 Note 1:</dt>
<dd>
    It might have been possible to implement a device abstraction that
    is compatible with \c fopen() but since this would have required
    to parse a string, and to take all the information needed either
    out of this string, or out of an additional table that would need to be
    provided by the application, this approach was not taken.
</dd>
<dt>\anchor stdio_note2 Note 2:</dt>
<dd>
    This basically follows the Unix approach: if a device such as a
    terminal needs special handling, it is in the domain of the
    terminal device driver to provide this functionality.  Thus, a
    simple function suitable as \c put() for \c fdevopen() that talks
    to a UART interface might look like this:

    \code
    int
    uart_putchar(char c, FILE *stream)
    {

      if (c == '\n')
        uart_putchar('\r', stream);
      loop_until_bit_is_set(UCSRA, UDRE);
      UDR = c;
      return 0;
    }
    \endcode
</dd>
<dt>\anchor stdio_note3 Note 3:</dt>
<dd>
    This implementation has been chosen because the cost of maintaining
    an alias is considerably smaller than the cost of maintaining full
    copies of each stream.  Yet, providing an implementation that offers
    the complete set of standard streams was deemed to be useful.  Not
    only that writing \c printf() instead of <tt>fprintf(mystream, ...)</tt>
    saves typing work, but since avr-gcc needs to resort to pass all
    arguments of variadic functions on the stack (as opposed to passing
    them in registers for functions that take a fixed number of
    parameters), the ability to pass one parameter less by implying
    \c stdin or stdout will also save some execution time.
</dd>
</dl>
*/

#if !defined(__DOXYGEN__)

/*
 * This is an internal structure of the library that is subject to be
 * changed without warnings at any time.  Please do *never* reference
 * elements of it beyond by using the official interfaces provided.
 */
struct __file {
    char    *buf;       /* buffer pointer */
    unsigned char unget;    /* ungetc() buffer */
    uint8_t  flags;      /* flags, see below */
#define __SRD   0x0001      /* OK to read */
#define __SWR   0x0002      /* OK to write */
#define __SSTR  0x0004      /* this is an sprintf/snprintf string */
#define __SPGM  0x0008      /* fmt string is in progmem */
#define __SERR  0x0010      /* found error */
#define __SEOF  0x0020      /* found EOF */
#define __SUNGET 0x040      /* ungetc() happened */
#define __SMALLOC 0x80      /* handle is malloc()ed */
#if 0
/* possible future extensions, will require uint16_t flags */
#define __SRW   0x0100      /* open for reading & writing */
#define __SLBF  0x0200      /* line buffered */
#define __SNBF  0x0400      /* unbuffered */
#define __SMBF  0x0800      /* buf is from malloc */
#endif
    int size;       /* size of buffer */
    int len;        /* characters read or written so far */
    int (*put)(char, struct __file *);  /* function to write one char to device */
    int (*get)(struct __file *);    /* function to read one char from device */
    void    *udata;     /* User defined and accessible data. */
```

```
};

#endif /* not __DOXYGEN__ */

/**@{*/
/**
   \c FILE is the opaque structure that is passed around between the
   various standard IO functions.
*/
typedef struct __file FILE;

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

   The first stream opened with read intent using \c fdevopen()
   will be assigned to \c stdin.
*/
#define stdin (__iob[0])

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

   The first stream opened with write intent using \c fdevopen()
   will be assigned to both, \c stdin, and \c stderr.
*/
#define stdout (__iob[1])

/**
   Stream destined for error output.  Unless specifically assigned,
   identical to \c stdout.

   If \c stderr should point to another stream, the result of
   another \c fdevopen() must be explicitly assigned to it without
   closing the previous \c stderr (since this would also close
   \c stdout).
*/
#define stderr (__iob[2])

/**
   \c EOF declares the value that is returned by various standard IO
   functions in case of an error.  Since the AVR platform (currently)
   doesn't contain an abstraction for actual files, its origin as
   "end of file" is somewhat meaningless here.
*/
#define EOF (-1)

/** This macro inserts a pointer to user defined data into a FILE
    stream object.

    The user data can be useful for tracking state in the put and get
    functions supplied to the fdevopen() function. */
#define fdev_set_udata(stream, u) do { (stream)->udata = u; } while(0)

/** This macro retrieves a pointer to user defined data from a FILE
    stream object. */
#define fdev_get_udata(stream) ((stream)->udata)

#if defined(__DOXYGEN__)
/**
   \brief Setup a user-supplied buffer as an stdio stream

   This macro takes a user-supplied buffer \c stream, and sets it up
   as a stream that is valid for stdio operations, similar to one that
   has been obtained dynamically from fdevopen(). The buffer to setup
   must be of type #FILE.

   The arguments \c put and \c get are identical to those that need to
   be passed to fdevopen().

   The \c rwflag argument can take one of the values #_FDEV_SETUP_READ,
   #_FDEV_SETUP_WRITE, or #_FDEV_SETUP_RW, for read, write, or read/write
   intent, respectively.

   \note No assignments to the standard streams will be performed by
   fdev_setup_stream().  If standard streams are to be used, these
   need to be assigned by the user.  See also under
   \ref stdio_without_malloc "Running stdio without malloc()".
 */
#define fdev_setup_stream(stream, put, get, rwflag)
#else  /* !DOXYGEN */
#define fdev_setup_stream(stream, p, g, f) \
    do { \
        (stream)->put = p; \
        (stream)->get = g; \
        (stream)->flags = f; \
```

```
        (stream)->udata = 0; \
    } while(0)
#endif /* DOXYGEN */

#define _FDEV_SETUP_READ  __SRD /**< fdev_setup_stream() with read intent */
#define _FDEV_SETUP_WRITE __SWR /**< fdev_setup_stream() with write intent */
#define _FDEV_SETUP_RW    (__SRD|__SWR) /**< fdev_setup_stream() with read/write intent */

/**
 * Return code for an error condition during device read.
 *
 * To be used in the get function of fdevopen().
 */
#define _FDEV_ERR (-1)

/**
 * Return code for an end-of-file condition during device read.
 *
 * To be used in the get function of fdevopen().
 */
#define _FDEV_EOF (-2)

#if defined(__DOXYGEN__)
/**
   \brief Initializer for a user-supplied stdio stream

   This macro acts similar to fdev_setup_stream(), but it is to be
   used as the initializer of a variable of type FILE.

   The remaining arguments are to be used as explained in
   fdev_setup_stream().
 */
#define FDEV_SETUP_STREAM(put, get, rwflag)
#else  /* !DOXYGEN */
/* In order to work with C++, we have to mention the fields in the order
   as they appear in struct __file.  Also, designated initializers are
   only supported since C++20.  */
#define FDEV_SETUP_STREAM(PU, GE, FL) \
    {                                 \
        (char*) 0  /* buf */,         \
        0u         /* unget */,       \
        FL         /* flags */,       \
        0          /* size */,        \
        0          /* len */,         \
        PU         /* put */,         \
        GE         /* get */,         \
        (void*) 0  /* udata */        \
    }
#endif /* DOXYGEN */

#ifdef __cplusplus
extern "C" {
#endif

#if !defined(__DOXYGEN__)
/*
 * Doxygen documentation can be found in fdevopen.c.
 */

extern struct __file *__iob[];

#if defined(__STDIO_FDEVOPEN_COMPAT_12)
/*
 * Declare prototype for the discontinued version of fdevopen() that
 * has been in use up to AVR-LibC 1.2.x.  The new implementation has
 * some backwards compatibility with the old version.
 */
extern FILE *fdevopen(int (*__put)(char), int (*__get)(void),
                      int __opts __attribute__((unused)));
#else  /* !defined(__STDIO_FDEVOPEN_COMPAT_12) */
/* New prototype for AVR-LibC 1.4 and above. */
extern FILE *fdevopen(int (*__put)(char, FILE*), int (*__get)(FILE*));
#endif /* defined(__STDIO_FDEVOPEN_COMPAT_12) */

#endif /* not __DOXYGEN__ */

/**
   This function closes \c stream, and disallows and further
   IO to and from it.

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

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

   It currently always returns 0 (for success).
```

```
*/
extern int  fclose(FILE *__stream);

/**
   This macro frees up any library resources that might be associated
   with \c stream.  It should be called if \c stream is no longer
   needed, right before the application is going to destroy the
   \c stream object itself.

   (Currently, this macro evaluates to nothing, but this might change
   in future versions of the library.)
*/
#if defined(__DOXYGEN__)
# define fdev_close()
#else
# define fdev_close() ((void)0)
#endif

/**
   \c vfprintf is the central facility of the \c printf family of
   functions.  It outputs values to \c stream under control of a
   format string passed in \c fmt.  The actual values to print are
   passed as a variable argument list \c ap.

   \c vfprintf returns the number of characters written to \c stream,
   or \c EOF in case of an error.  Currently, this will only happen
   if \c stream has not been opened with write intent.

   The format string is composed of zero or more directives: ordinary
   characters (not \c %), which are copied unchanged to the output
   stream; and conversion specifications, each of which results in
   fetching zero or more subsequent arguments.  Each conversion
   specification is introduced by the \c % character.  The arguments must
   properly correspond (after type promotion) with the conversion
   specifier.  After the \c %, the following appear in sequence:

   - Zero or more of the following flags:
      <ul>
      <li> \c # The value should be converted to an "alternate form".  For
            c, d, i, s, and u conversions, this option has no effect.
            For o conversions, the precision of the number is
            increased to force the first character of the output
            string to a zero (except if a zero value is printed with
            an explicit precision of zero).  For x and X conversions,
            a non-zero result has the string `0x' (or `0X' for X
            conversions) prepended to it.</li>
      <li> \c 0 (zero) Zero padding.  For all conversions, the converted
            value is padded on the left with zeros rather than blanks.
            If a precision is given with a numeric conversion (d, i,
            o, u, i, x, and X), the 0 flag is ignored.</li>
      <li> \c - A negative field width flag; the converted value is to be
            left adjusted on the field boundary.  The converted value
            is padded on the right with blanks, rather than on the
            left with blanks or zeros.  A - overrides a 0 if both are
            given.</li>
      <li> ' ' (space) A blank should be left before a positive number
            produced by a signed conversion (d, or i).</li>
      <li> \c + A sign must always be placed before a number produced by a
            signed conversion.  A + overrides a space if both are
            used.</li>
      </ul>

   -   An optional decimal digit string specifying a minimum field width.
       If the converted value has fewer characters than the field width, it
       will be padded with spaces on the left (or right, if the left-adjustment
       flag has been given) to fill out the field width.
   -   An optional precision, in the form of a period . followed by an
       optional digit string.  If the digit string is omitted, the
       precision is taken as zero.  This gives the minimum number of
       digits to appear for d, i, o, u, x, and X conversions, or the
       maximum number of characters to be printed from a string for \c s
       conversions.
   -   An optional \c l or \c h length modifier, that specifies that the
       argument for the d, i, o, u, x, or X conversion is a \c "long int"
       rather than \c int. The \c h is ignored, as \c "short int" is
       equivalent to \c int.
   -   A character that specifies the type of conversion to be applied.

   The conversion specifiers and their meanings are:

   - \c diouxX The int (or appropriate variant) argument is converted
           to signed decimal (d and i), unsigned octal (o), unsigned
           decimal (u), or unsigned hexadecimal (x and X) notation.
           The letters "abcdef" are used for x conversions; the
           letters "ABCDEF" are used for X conversions.  The
           precision, if any, gives the minimum number of digits that
           must appear; if the converted value requires fewer digits,
```

```
        it is padded on the left with zeros.
- \c p  The <tt>void *</tt> argument is taken as an unsigned integer,
        and converted similarly as a <tt>%\#x</tt> command would do.
- \c c  The \c int argument is converted to an \c "unsigned char", and the
        resulting character is written.
- \c s  The \c "char *" argument is expected to be a pointer to an array
        of character type (pointer to a string).  Characters from
        the array are written up to (but not including) a
        terminating NUL character; if a precision is specified, no
        more than the number specified are written.  If a precision
        is given, no null character need be present; if the
        precision is not specified, or is greater than the size of
        the array, the array must contain a terminating NUL
        character.
- \c %  A \c % is written.  No argument is converted.  The complete
        conversion specification is "%%".
- \c eE The double argument is rounded and converted in the format
        \c "[-]d.ddde±dd" where there is one digit before the
        decimal-point character and the number of digits after it
        is equal to the precision; if the precision is missing, it
        is taken as 6; if the precision is zero, no decimal-point
        character appears.  An \e E conversion uses the letter \c 'E'
        (rather than \c 'e') to introduce the exponent.  The exponent
        always contains two digits; if the value is zero,
        the exponent is 00.
- \c fF The double argument is rounded and converted to decimal notation
        in the format \c "[-]ddd.ddd", where the number of digits after the
        decimal-point character is equal to the precision specification.
        If the precision is missing, it is taken as 6; if the precision
        is explicitly zero, no decimal-point character appears.  If a
        decimal point appears, at least one digit appears before it.
- \c gG The double argument is converted in style \c f or \c e (or
        \c F or \c E for \c G conversions).  The precision
        specifies the number of significant digits.  If the
        precision is missing, 6 digits are given; if the precision
        is zero, it is treated as 1.  Style \c e is used if the
        exponent from its conversion is less than -4 or greater
        than or equal to the precision.  Trailing zeros are removed
        from the fractional part of the result; a decimal point
        appears only if it is followed by at least one digit.
- \c S  Similar to the \c s format, except the pointer is expected to
        point to a program-memory (ROM) string instead of a RAM string.

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

Since the full implementation of all the mentioned features becomes
fairly large, three different flavours of vfprintf() can be
selected using linker options.  The default vfprintf() implements
all the mentioned functionality except floating point conversions.
A minimized version of vfprintf() is available that only implements
the very basic integer and string conversion facilities, but only
the \c # additional option can be specified using conversion
flags (these flags are parsed correctly from the format
specification, but then simply ignored).  This version can be
requested using the following \ref gcc_minusW "compiler options":

\code
-Wl,-u,vfprintf -lprintf_min
\endcode

If the full functionality including the floating point conversions
is required, the following options should be used:

\code
-Wl,-u,vfprintf -lprintf_flt -lm
\endcode

\par Limitations:
- The specified width and precision can be at most 255.

\par Notes:
- For floating-point conversions, if you link default or minimized
  version of vfprintf(), the symbol \c ? will be output and double
  argument will be skipped. So you output below will not be crashed.
  For default version the width field and the "pad to left" ( symbol
  minus ) option will work in this case.
- The \c hh length modifier is ignored (\c char argument is
  promoted to \c int). More exactly, this realization does not check
  the number of \c h symbols.
- But the \c ll length modifier will to abort the output, as this
  realization does not operate \c long \c long arguments.
- The variable width or precision field (an asterisk \c * symbol)
  is not realized and will to abort the output.

*/
```

```
extern int  vfprintf(FILE *__stream, const char *__fmt, va_list __ap);

/**
   Variant of \c vfprintf() that uses a \c fmt string that resides
   in program memory.
*/
extern int  vfprintf_P(FILE *__stream, const char *__fmt, va_list __ap);

/**
   The function \c fputc sends the character \c c (though given as type
   \c int) to \c stream.  It returns the character, or \c EOF in case
   an error occurred.
*/
extern int  fputc(int __c, FILE *__stream);

#if !defined(__DOXYGEN__)

/* putc() function implementation, required by standard */
extern int  putc(int __c, FILE *__stream);

/* putchar() function implementation, required by standard */
extern int  putchar(int __c);

#endif /* not __DOXYGEN__ */

/**
   The macro \c putc used to be a "fast" macro implementation with a
   functionality identical to fputc().  For space constraints, in
   AVR-LibC, it is just an alias for \c fputc.
*/
#define putc(__c, __stream) fputc(__c, __stream)

/**
   The macro \c putchar sends character \c c to \c stdout.
*/
#define putchar(__c) fputc(__c, stdout)

/**
   The function \c printf performs formatted output to stream
   \c stdout.  See \c vfprintf() for details.
*/
extern int  printf(const char *__fmt, ...);

/**
   Variant of \c printf() that uses a \c fmt string that resides
   in program memory.
*/
extern int  printf_P(const char *__fmt, ...);

/**
   The function \c vprintf performs formatted output to stream
   \c stdout, taking a variable argument list as in vfprintf().

   See vfprintf() for details.
*/
extern int  vprintf(const char *__fmt, va_list __ap);

/**
   Variant of \c printf() that sends the formatted characters
   to string \c s.
*/
extern int  sprintf(char *__s, const char *__fmt, ...);

/**
   Variant of \c sprintf() that uses a \c fmt string that resides
   in program memory.
*/
extern int  sprintf_P(char *__s, const char *__fmt, ...);

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

   Returns the number of characters that would have been written to
   \c s if there were enough space.
*/
extern int  snprintf(char *__s, size_t __n, const char *__fmt, ...);

/**
   Variant of \c snprintf() that uses a \c fmt string that resides
   in program memory.
*/
extern int  snprintf_P(char *__s, size_t __n, const char *__fmt, ...);

/**
```

```
   Like \c sprintf() but takes a variable argument list for the
   arguments.
*/
extern int  vsprintf(char *__s, const char *__fmt, va_list ap);

/**
   Variant of \c vsprintf() that uses a \c fmt string that resides
   in program memory.
*/
extern int  vsprintf_P(char *__s, const char *__fmt, va_list ap);

/**
   Like \c vsprintf(), but instead of assuming \c s to be of infinite
   size, no more than \c n characters (including the trailing NUL
   character) will be converted to \c s.

   Returns the number of characters that would have been written to
   \c s if there were enough space.
*/
extern int  vsnprintf(char *__s, size_t __n, const char *__fmt, va_list ap);

/**
   Variant of \c vsnprintf() that uses a \c fmt string that resides
   in program memory.
*/
extern int  vsnprintf_P(char *__s, size_t __n, const char *__fmt, va_list ap);
/**
   The function \c fprintf performs formatted output to \c stream.
   See \c vfprintf() for details.
*/
extern int  fprintf(FILE *__stream, const char *__fmt, ...);

/**
   Variant of \c fprintf() that uses a \c fmt string that resides
   in program memory.
*/
extern int  fprintf_P(FILE *__stream, const char *__fmt, ...);

/**
   Write the string pointed to by \c str to stream \c stream.

   Returns 0 on success and EOF on error.
*/
extern int  fputs(const char *__str, FILE *__stream);

/**
   Variant of fputs() where \c str resides in program memory.
*/
extern int  fputs_P(const char *__str, FILE *__stream);

/**
   Write the string pointed to by \c str, and a trailing newline
   character, to \c stdout.
*/
extern int  puts(const char *__str);

/**
   Variant of puts() where \c str resides in program memory.
*/
extern int  puts_P(const char *__str);

/**
   Write \c nmemb objects, \c size bytes each, to \c stream.
   The first byte of the first object is referenced by \c ptr.

   Returns the number of objects successfully written, i. e.
   \c nmemb unless an output error occured.
 */
extern size_t   fwrite(const void *__ptr, size_t __size, size_t __nmemb,
                FILE *__stream);

/**
   The function \c fgetc reads a character from \c stream.  It returns
   the character, or \c EOF in case end-of-file was encountered or an
   error occurred.  The routines feof() or ferror() must be used to
   distinguish between both situations.
*/
extern int  fgetc(FILE *__stream);

#if !defined(__DOXYGEN__)

/* getc() function implementation, required by standard */
extern int  getc(FILE *__stream);

/* getchar() function implementation, required by standard */
extern int  getchar(void);

```

```
#endif /* not __DOXYGEN__ */

/**
   The macro \c getc used to be a "fast" macro implementation with a
   functionality identical to fgetc().  For space constraints, in
   AVR-LibC, it is just an alias for \c fgetc.
*/
#define getc(__stream) fgetc(__stream)

/**
   The macro \c getchar reads a character from \c stdin.  Return
   values and error handling is identical to fgetc().
*/
#define getchar() fgetc(stdin)

/**
   The ungetc() function pushes the character \c c (converted to an
   unsigned char) back onto the input stream pointed to by \c stream.
   The pushed-back character will be returned by a subsequent read on
   the stream.

   Currently, only a single character can be pushed back onto the
   stream.

   The ungetc() function returns the character pushed back after the
   conversion, or \c EOF if the operation fails.  If the value of the
   argument \c c character equals \c EOF, the operation will fail and
   the stream will remain unchanged.
*/
extern int  ungetc(int __c, FILE *__stream);

/**
   Read at most <tt>size - 1</tt> bytes from \c stream, until a
   newline character was encountered, and store the characters in the
   buffer pointed to by \c str.  Unless an error was encountered while
   reading, the string will then be terminated with a \c NUL
   character.

   If an error was encountered, the function returns NULL and sets the
   error flag of \c stream, which can be tested using ferror().
   Otherwise, a pointer to the string will be returned.  */
extern char *fgets(char *__str, int __size, FILE *__stream);

/**
   Similar to fgets() except that it will operate on stream \c stdin,
   and the trailing newline (if any) will not be stored in the string.
   It is the caller's responsibility to provide enough storage to hold
   the characters read.  */
extern char *gets(char *__str);

/**
   Read \c nmemb objects, \c size bytes each, from \c stream,
   to the buffer pointed to by \c ptr.

   Returns the number of objects successfully read, i. e.
   \c nmemb unless an input error occured or end-of-file was
   encountered.  feof() and ferror() must be used to distinguish
   between these two conditions.
 */
extern size_t   fread(void *__ptr, size_t __size, size_t __nmemb,
               FILE *__stream);

/**
   Clear the error and end-of-file flags of \c stream.
 */
extern void clearerr(FILE *__stream);

#if !defined(__DOXYGEN__)
/* fast inlined version of clearerr() */
#define clearerror(s) do { (s)->flags &= ~(__SERR | __SEOF); } while(0)
#endif /* !defined(__DOXYGEN__) */

/**
   Test the end-of-file flag of \c stream.  This flag can only be cleared
   by a call to clearerr().
 */
extern int  feof(FILE *__stream);

#if !defined(__DOXYGEN__)
/* fast inlined version of feof() */
#define feof(s) ((s)->flags & __SEOF)
#endif /* !defined(__DOXYGEN__) */

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

```c
extern int  ferror(FILE *__stream);

#if !defined(__DOXYGEN__)
/* fast inlined version of ferror() */
#define ferror(s) ((s)->flags & __SERR)
#endif /* !defined(__DOXYGEN__) */

extern int  vfscanf(FILE *__stream, const char *__fmt, va_list __ap);

/**
   Variant of vfscanf() using a \c fmt string in program memory.
 */
extern int  vfscanf_P(FILE *__stream, const char *__fmt, va_list __ap);

/**
   The function \c fscanf performs formatted input, reading the
   input data from \c stream.

   See vfscanf() for details.
 */
extern int  fscanf(FILE *__stream, const char *__fmt, ...);

/**
   Variant of fscanf() using a \c fmt string in program memory.
 */
extern int  fscanf_P(FILE *__stream, const char *__fmt, ...);

/**
   The function \c scanf performs formatted input from stream \c stdin.

   See vfscanf() for details.
 */
extern int  scanf(const char *__fmt, ...);

/**
   Variant of scanf() where \c fmt resides in program memory.
 */
extern int  scanf_P(const char *__fmt, ...);

/**
   The function \c vscanf performs formatted input from stream
   \c stdin, taking a variable argument list as in vfscanf().

   See vfscanf() for details.
*/
extern int  vscanf(const char *__fmt, va_list __ap);

/**
   The function \c sscanf performs formatted input, reading the
   input data from the buffer pointed to by \c buf.

   See vfscanf() for details.
 */
extern int  sscanf(const char *__buf, const char *__fmt, ...);

/**
   Variant of sscanf() using a \c fmt string in program memory.
 */
extern int  sscanf_P(const char *__buf, const char *__fmt, ...);

#if defined(__DOXYGEN__)
/**
   Flush \c stream.

   This is a null operation provided for source-code compatibility
   only, as the standard IO implementation currently does not perform
   any buffering.
 */
extern int  fflush(FILE *stream);
#else
static __inline__ int fflush(FILE *stream __attribute__((unused)))
{
    return 0;
}
#endif

#ifndef __DOXYGEN__
/* only mentioned for libstdc++ support, not implemented in library */
#define BUFSIZ 1024
#define _IONBF 0
__extension__ typedef long long fpos_t;
extern int fgetpos(FILE *stream, fpos_t *pos);
extern FILE *fopen(const char *path, const char *mode);
extern FILE *freopen(const char *path, const char *mode, FILE *stream);
extern FILE *fdopen(int, const char *);
extern int fseek(FILE *stream, long offset, int whence);
extern int fsetpos(FILE *stream, fpos_t *pos);
```

```
extern long ftell(FILE *stream);
extern int fileno(FILE *);
extern void perror(const char *s);
extern int remove(const char *pathname);
extern int rename(const char *oldpath, const char *newpath);
extern void rewind(FILE *stream);
extern void setbuf(FILE *stream, char *buf);
extern int setvbuf(FILE *stream, char *buf, int mode, size_t size);
extern FILE *tmpfile(void);
extern char *tmpnam (char *s);
#endif  /* !__DOXYGEN__ */

#ifdef __cplusplus
}
#endif

/**@}*/

#ifndef __DOXYGEN__
/*
 * The following constants are currently not used by AVR-LibC's
 * stdio subsystem.  They are defined here since the gcc build
 * environment expects them to be here.
 */
#define SEEK_SET 0
#define SEEK_CUR 1
#define SEEK_END 2

#endif

#endif /* __ASSEMBLER */

#endif /* _STDIO_H_ */
```

23.56 stdlib.h File Reference

Data Structures

- struct div_t
- struct ldiv_t

Macros

- #define RAND_MAX 0x7FFF

Typedefs

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

Functions

- void abort (void)
- int abs (int __i)
- long labs (long __i)
- void ∗ bsearch (const void ∗__key, const void ∗__base, size_t __nmemb, size_t __size, int(∗__compar)(const void ∗, const void ∗))
- div_t div (int __num, int __denom) __asm__("__divmodhi4")
- ldiv_t ldiv (long __num, long __denom) __asm__("__divmodsi4")
- void qsort (void ∗__base, size_t __nmemb, size_t __size, __compar_fn_t __compar)
- long strtol (const char ∗__nptr, char ∗∗__endptr, int __base)
- unsigned long strtoul (const char ∗__nptr, char ∗∗__endptr, int __base)
- long atol (const char ∗__s)
- int atoi (const char ∗__s)

- void exit (int __status)
- void ∗ malloc (size_t __size)
- void free (void ∗__ptr)
- void ∗ calloc (size_t __nele, size_t __size)
- void ∗ realloc (void ∗__ptr, size_t __size)
- float strtof (const char ∗__nptr, char ∗∗__endptr)
- double strtod (const char ∗__nptr, char ∗∗__endptr)
- long double strtold (const char ∗__nptr, char ∗∗__endptr)
- int atexit (void(∗func)(void))
- float atoff (const char ∗__nptr)
- double atof (const char ∗__nptr)
- long double atofl (const char ∗__nptr)
- int rand (void)
- void srand (unsigned int __seed)
- int rand_r (unsigned long ∗__ctx)

Variables

- size_t __malloc_margin
- char ∗ __malloc_heap_start
- char ∗ __malloc_heap_end

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

- #define RANDOM_MAX 0x7FFFFFFF
- char ∗ itoa (int val, char ∗s, int radix)
- char ∗ ltoa (long val, char ∗s, int radix)
- char ∗ utoa (unsigned int val, char ∗s, int radix)
- char ∗ ultoa (unsigned long val, char ∗s, int radix)
- long random (void)
- void srandom (unsigned long __seed)
- long random_r (unsigned long ∗__ctx)

Conversion functions for double arguments.

- #define DTOSTR_ALWAYS_SIGN 0x01 /∗ put '+' or ' ' for positives ∗/
- #define DTOSTR_PLUS_SIGN 0x02 /∗ put '+' rather than ' ' ∗/
- #define DTOSTR_UPPERCASE 0x04 /∗ put 'E' rather 'e' ∗/
- #define EXIT_SUCCESS 0
- #define EXIT_FAILURE 1
- char ∗ ftostre (float __val, char ∗__s, unsigned char __prec, unsigned char __flags)
- char ∗ dtostre (double __val, char ∗__s, unsigned char __prec, unsigned char __flags)
- char ∗ ldtostre (long double __val, char ∗__s, unsigned char __prec, unsigned char __flags)
- char ∗ ftostrf (float __val, signed char __width, unsigned char __prec, char ∗__s)
- char ∗ dtostrf (double __val, signed char __width, unsigned char __prec, char ∗__s)
- char ∗ ldtostrf (long double __val, signed char __width, unsigned char __prec, char ∗__s)

23.57 stdlib.h

Go to the documentation of this file.

```
/* Copyright (c) 2002, Marek Michalkiewicz
   Copyright (c) 2004,2007 Joerg Wunsch

   Portions of documentation Copyright (c) 1990, 1991, 1993, 1994
   The Regents of the University of California.

   All rights reserved.

   Redistribution and use in source and binary forms, with or without
   modification, are permitted provided that the following conditions are met:

   * Redistributions of source code must retain the above copyright
     notice, this list of conditions and the following disclaimer.

   * Redistributions in binary form must reproduce the above copyright
     notice, this list of conditions and the following disclaimer in
     the documentation and/or other materials provided with the
     distribution.

   * Neither the name of the copyright holders nor the names of
     contributors may be used to endorse or promote products derived
     from this software without specific prior written permission.

  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
  AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
  IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
  ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
  LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
  CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
  SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
  INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
  CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
  ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
  POSSIBILITY OF SUCH DAMAGE.

  $Id$
*/

#ifndef _STDLIB_H_
#define _STDLIB_H_ 1

#ifndef __ASSEMBLER__

#ifndef __DOXYGEN__
#define __need_NULL
#define __need_size_t
#define __need_wchar_t
#include <stddef.h>

#ifndef __ptr_t
#define __ptr_t void *
#endif
#endif  /* !__DOXYGEN__ */

#ifdef __cplusplus
extern "C" {
#endif

/** \file */

/** \defgroup avr_stdlib <stdlib.h>: General utilities
    \code #include <stdlib.h> \endcode

    This file declares some basic C macros and functions as
    defined by the ISO standard, plus some AVR-specific extensions.
*/

/** \ingroup avr_stdlib */
/**@{*/
/** Result type for function div(). */
typedef struct {
    int quot;                   /**< The Quotient. */
    int rem;                    /**< The Remainder. */
} div_t;

/** Result type for function ldiv(). */
typedef struct {
    long quot;                  /**< The Quotient. */
    long rem;                   /**< The Remainder. */
} ldiv_t;

/** Comparision function type for qsort(), just for convenience. */
typedef int (*__compar_fn_t)(const void *, const void *);
```

```
#ifndef __DOXYGEN__

#ifndef __ATTR_CONST__
# define __ATTR_CONST__ __attribute__((__const__))
#endif

#ifndef __ATTR_MALLOC__
# define __ATTR_MALLOC__ __attribute__((__malloc__))
#endif

#ifndef __ATTR_NORETURN__
# define __ATTR_NORETURN__ __attribute__((__noreturn__))
#endif

#ifndef __ATTR_PURE__
# define __ATTR_PURE__ __attribute__((__pure__))
#endif

#ifndef __ATTR_GNU_INLINE__
# ifdef  __GNUC_STDC_INLINE__
#  define __ATTR_GNU_INLINE__   __attribute__((__gnu_inline__))
# else
#  define __ATTR_GNU_INLINE__
# endif
#endif

#ifndef __ATTR_ALWAYS_INLINE__
#define __ATTR_ALWAYS_INLINE__ __inline__ __attribute__((__always_inline__))
#endif

#endif

/** The abort() function causes abnormal program termination to occur.
    This realization disables interrupts and jumps to _exit() function
    with argument equal to 1. In the limited AVR environment, execution is
    effectively halted by entering an infinite loop. */
extern void abort(void) __ATTR_NORETURN__;

#ifndef __DOXYGEN__
static __ATTR_ALWAYS_INLINE__
int abs (int __i)
{
    return __builtin_abs (__i);
}
#endif
/** The abs() function computes the absolute value of the integer \c i.
   \note The abs() and labs() functions are builtins of gcc.
*/
extern int abs(int __i) __ATTR_CONST__;

#ifndef __DOXYGEN__
static __ATTR_ALWAYS_INLINE__
long labs (long __i)
{
    return __builtin_labs (__i);
}
#endif
/** The labs() function computes the absolute value of the long integer
    \c i.
   \note The abs() and labs() functions are builtins of gcc.
*/
extern long labs(long __i) __ATTR_CONST__;

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

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

     The bsearch() function returns a pointer to a matching member of
     the array, or a null pointer if no match is found.  If two
     members compare as equal, which member is matched is unspecified.
*/
extern void *bsearch(const void *__key, const void *__base, size_t __nmemb,
             size_t __size, int (*__compar)(const void *, const void *));

/* __divmodhi4 and __divmodsi4 from libgcc.a */
/**
```

```
     The div() function computes the value \c num/denom and returns
     the quotient and remainder in a structure named \c div_t that
     contains two int members named \c quot and \c rem.
*/
extern div_t div(int __num, int __denom) __asm__("__divmodhi4") __ATTR_CONST__;
/**
     The ldiv() function computes the value \c num/denom and returns
     the quotient and remainder in a structure named \c ldiv_t that
     contains two long integer members named \c quot and \c rem.
*/
extern ldiv_t ldiv(long __num, long __denom) __asm__("__divmodsi4") __ATTR_CONST__;

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

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

     The comparison function must return an integer less than, equal
     to, or greater than zero if the first argument is considered to
     be respectively less than, equal to, or greater than the second.
*/
extern void qsort(void *__base, size_t __nmemb, size_t __size,
          __compar_fn_t __compar);

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

    The string may begin with an arbitrary amount of white space (as
    determined by isspace()) followed by a single optional \c '+' or \c '-'
    sign.  If \c base is zero or 16, the string may then include a
    \c "0x" prefix, and the number will be read in base 16; otherwise,
    a zero base is taken as 10 (decimal) unless the next character is
    \c '0', in which case it is taken as 8 (octal).

    The remainder of the string is converted to a long value in the
    obvious manner, stopping at the first character which is not a
    valid digit in the given base.  (In bases above 10, the letter \c 'A'
    in either upper or lower case represents 10, \c 'B' represents 11,
    and so forth, with \c 'Z' representing 35.)

    If \c endptr is not NULL, strtol() stores the address of the first
    invalid character in \c *endptr.  If there were no digits at all,
    however, strtol() stores the original value of \c nptr in \c
    *endptr.  (Thus, if \c *nptr is not \c '\\0' but \c **endptr is \c '\\0'
    on return, the entire string was valid.)

    The strtol() function returns the result of the conversion, unless
    the value would underflow or overflow.  If no conversion could be
    performed, 0 is returned.  If an overflow or underflow occurs, \c
    errno is set to \ref avr_errno "ERANGE" and the function return value
    is clamped to \c LONG_MIN or \c LONG_MAX, respectively.
*/
extern long strtol(const char *__nptr, char **__endptr, int __base);

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

    The string may begin with an arbitrary amount of white space (as
    determined by isspace()) followed by a single optional \c '+' or \c '-'
    sign.  If \c base is zero or 16, the string may then include a
    \c "0x" prefix, and the number will be read in base 16; otherwise,
    a zero base is taken as 10 (decimal) unless the next character is
    \c '0', in which case it is taken as 8 (octal).

    The remainder of the string is converted to an unsigned long value
    in the obvious manner, stopping at the first character which is
    not a valid digit in the given base.  (In bases above 10, the
    letter \c 'A' in either upper or lower case represents 10, \c 'B'
    represents 11, and so forth, with \c 'Z' representing 35.)

    If \c endptr is not NULL, strtoul() stores the address of the first
    invalid character in \c *endptr.  If there were no digits at all,
    however, strtoul() stores the original value of \c nptr in \c
    *endptr.  (Thus, if \c *nptr is not \c '\\0' but \c **endptr is \c '\\0'
    on return, the entire string was valid.)

```

```
    The strtoul() function return either the result of the conversion
    or, if there was a leading minus sign, the negation of the result
    of the conversion, unless the original (non-negated) value would
    overflow; in the latter case, strtoul() returns ULONG_MAX, and \c
    errno is set to \ref avr_errno "ERANGE".  If no conversion could
    be performed, 0 is returned.
*/
extern unsigned long strtoul(const char *__nptr, char **__endptr, int __base);

/**
    The atol() function converts the initial portion of the string
    pointed to by \p s to long integer representation. In contrast to

        \code strtol(s, (char **)NULL, 10); \endcode

    this function does not detect overflow (\c errno is not changed and
    the result value is not predictable), uses smaller memory (flash and
    stack) and works more quickly.
*/
extern long atol(const char *__s) __ATTR_PURE__;

/**
    The atoi() function converts the initial portion of the string
    pointed to by \p s to integer representation. In contrast to

        \code (int)strtol(s, (char **)NULL, 10); \endcode

    this function does not detect overflow (\c errno is not changed and
    the result value is not predictable), uses smaller memory (flash and
    stack) and works more quickly.
*/
extern int atoi(const char *__s) __ATTR_PURE__;

/**
   The exit() function terminates the application.  Since there is no
   environment to return to, \c status is ignored, and code execution
   will eventually reach an infinite loop, thereby effectively halting
   all code processing.  Before entering the infinite loop, interrupts
   are globally disabled.

   Global destructors will be called before halting
   execution, see the \ref sec_dot_fini ".fini" sections.
*/
extern void exit(int __status) __ATTR_NORETURN__;

/**
   The malloc() function allocates \c size bytes of memory.
   If malloc() fails, a NULL pointer is returned.

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

   See the chapter about \ref malloc "malloc() usage" for implementation
   details.
*/
extern void *malloc(size_t __size) __ATTR_MALLOC__;

/**
   The free() function causes the allocated memory referenced by \c
   ptr to be made available for future allocations.  If \c ptr is
   NULL, no action occurs.
*/
extern void free(void *__ptr);

/**
   \c malloc() \ref malloc_tunables "tunable".
*/
extern size_t __malloc_margin;

/**
   \c malloc() \ref malloc_tunables "tunable".
*/
extern char *__malloc_heap_start;

/**
   \c malloc() \ref malloc_tunables "tunable".
*/
extern char *__malloc_heap_end;

/**
   Allocate \c nele elements of \c size each.  Identical to calling
   \c malloc() using <tt>nele * size</tt> as argument, except the
   allocated memory will be cleared to zero.
*/
extern void *calloc(size_t __nele, size_t __size) __ATTR_MALLOC__;

/**
```

```
   The realloc() function tries to change the size of the region
   allocated at \c ptr to the new \c size value.  It returns a
   pointer to the new region.  The returned pointer might be the
   same as the old pointer, or a pointer to a completely different
   region.

   The contents of the returned region up to either the old or the new
   size value (whatever is less) will be identical to the contents of
   the old region, even in case a new region had to be allocated.

   It is acceptable to pass \c ptr as NULL, in which case realloc()
   will behave identical to malloc().

   If the new memory cannot be allocated, realloc() returns NULL, and
   the region at \c ptr will not be changed.
*/
extern void *realloc(void *__ptr, size_t __size) __ATTR_MALLOC__;

extern float strtof(const char *__nptr, char **__endptr);
/** \ingroup avr_stdlib
    The strtod() function is similar to strtof(), except that the conversion
    result is of type \c double instead of \c float.

    strtod() is currently only supported when \c double is a 32-bit type. */
extern double strtod(const char *__nptr, char **__endptr);
/** \ingroup avr_stdlib
    The strtold() function is similar to strtof(), except that the conversion
    result is of type \c long \c double instead of \c float.

    strtold() is currently only supported when \c long \c double is a
    32-bit type. */
extern long double strtold(const char *__nptr, char **__endptr);

/**
   \ingroup avr_stdlib
   The atexit() function registers function \a func to be run as part of
   the \c exit() function during \ref sec_dot_fini ".fini8".
   atexit() calls malloc().
 */
extern int atexit(void (*func)(void));

/** \ingroup avr_stdlib
    \fn float atoff (const char *nptr)

    The atoff() function converts the initial portion of the string pointed
    to by \a nptr to \c float representation.

    It is equivalent to calling
    \code strtof(nptr, (char**) 0); \endcode */
extern float atoff(const char *__nptr);
/** \ingroup avr_stdlib
    \fn  double atof (const char *nptr)

    The atof() function converts the initial portion of the string pointed
    to by \a nptr to \c double representation.

    It is equivalent to calling
    \code strtod(nptr, (char**) 0); \endcode */
extern double atof(const char *__nptr);
/** \ingroup avr_stdlib
    \fn long double atofl (const char *nptr)

    The atofl() function converts the initial portion of the string pointed
    to by \a nptr to \c long \c double representation.

    It is equivalent to calling
    \code strtold(nptr, (char**) 0); \endcode */
extern long double atofl(const char *__nptr);

/** Highest number that can be generated by rand(). */
#define RAND_MAX 0x7FFF

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

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

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

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

```
     32-bit precision.
*/
extern int rand(void);
/**
   Pseudo-random number generator seeding; see rand().
*/
extern void srand(unsigned int __seed);

/**
   Variant of rand() that stores the context in the user-supplied
   variable located at \c ctx instead of a static library variable
   so the function becomes re-entrant.
*/
extern int rand_r(unsigned long *__ctx);
/**@}*/

/**@{*/
/** \name Non-standard (i.e. non-ISO C) functions.
 \ingroup avr_stdlib
*/
/**
   \brief Convert an integer to a string.

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

   \note The minimal size of the buffer \c s depends on the choice of
   radix. For example, if the radix is 2 (binary), you need to supply a buffer
   with a minimal length of 8 * sizeof (int) + 1 characters, i.e. one
   character for each bit plus one for the string terminator. Using a larger
   radix will require a smaller minimal buffer size.

   \warning If the buffer is too small, you risk a buffer overflow.

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

    If radix is 10 and val is negative, a minus sign will be prepended.

   The itoa() function returns the pointer passed as \c s.
*/
#ifdef  __DOXYGEN__
extern char *itoa(int val, char *s, int radix);
#else
extern __inline__ __ATTR_GNU_INLINE__
char *itoa (int __val, char *__s, int __radix)
{
    if (!__builtin_constant_p (__radix)) {
    extern char *__itoa (int, char *, int);
    return __itoa (__val, __s, __radix);
    } else if (__radix < 2 || __radix > 36) {
    *__s = 0;
    return __s;
    } else {
    extern char *__itoa_ncheck (int, char *, unsigned char);
    return __itoa_ncheck (__val, __s, __radix);
    }
}
#endif

/**
 \ingroup avr_stdlib

   \brief Convert a long integer to a string.

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

   \note The minimal size of the buffer \c s depends on the choice of
   radix. For example, if the radix is 2 (binary), you need to supply a buffer
   with a minimal length of 8 * sizeof (long int) + 1 characters, i.e. one
   character for each bit plus one for the string terminator. Using a larger
   radix will require a smaller minimal buffer size.

   \warning If the buffer is too small, you risk a buffer overflow.

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

   If radix is 10 and val is negative, a minus sign will be prepended.

```

```
   The ltoa() function returns the pointer passed as \c s.
*/
#ifdef  __DOXYGEN__
extern char *ltoa(long val, char *s, int radix);
#else
extern __inline__ __ATTR_GNU_INLINE__
char *ltoa (long __val, char *__s, int __radix)
{
    if (!__builtin_constant_p (__radix))
    {
    extern char *__ltoa (long, char *, int);
    return __ltoa (__val, __s, __radix);
    }
    else if (__radix < 2 || __radix > 36)
    {
    *__s = 0;
    return __s;
    }
    else
    {
    extern char *__ltoa_ncheck (long, char *, unsigned char);
    return __ltoa_ncheck (__val, __s, __radix);
    }
}
#endif

/**
 \ingroup avr_stdlib

   \brief Convert an unsigned integer to a string.

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

   \note The minimal size of the buffer \c s depends on the choice of
   radix. For example, if the radix is 2 (binary), you need to supply a buffer
   with a minimal length of 8 * sizeof (unsigned int) + 1 characters, i.e. one
   character for each bit plus one for the string terminator. Using a larger
   radix will require a smaller minimal buffer size.

   \warning If the buffer is too small, you risk a buffer overflow.

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

   The utoa() function returns the pointer passed as \c s.
*/
#ifdef  __DOXYGEN__
extern char *utoa(unsigned int val, char *s, int radix);
#else
extern __inline__ __ATTR_GNU_INLINE__
char *utoa (unsigned int __val, char *__s, int __radix)
{
    if (!__builtin_constant_p (__radix))
    {
    extern char *__utoa (unsigned int, char *, int);
    return __utoa (__val, __s, __radix);
    }
    else if (__radix < 2 || __radix > 36)
    {
    *__s = 0;
    return __s;
    }
    else
    {
    extern char *__utoa_ncheck (unsigned int, char *, unsigned char);
    return __utoa_ncheck (__val, __s, __radix);
    }
}
#endif

/**
 \ingroup avr_stdlib
   \brief Convert an unsigned long integer to a string.

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

   \note The minimal size of the buffer \c s depends on the choice of
   radix. For example, if the radix is 2 (binary), you need to supply a buffer
   with a minimal length of 8 * sizeof (unsigned long int) + 1 characters,
   i.e. one character for each bit plus one for the string terminator. Using a
   larger radix will require a smaller minimal buffer size.
```

```

   \warning If the buffer is too small, you risk a buffer overflow.

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

   The ultoa() function returns the pointer passed as \c s.
*/
#ifdef  __DOXYGEN__
extern char *ultoa(unsigned long val, char *s, int radix);
#else
extern __inline__ __ATTR_GNU_INLINE__
char *ultoa (unsigned long __val, char *__s, int __radix)
{
    if (!__builtin_constant_p (__radix)) {
    extern char *__ultoa (unsigned long, char *, int);
    return __ultoa (__val, __s, __radix);
    } else if (__radix < 2 || __radix > 36) {
    *__s = 0;
    return __s;
    } else {
    extern char *__ultoa_ncheck (unsigned long, char *, unsigned char);
    return __ultoa_ncheck (__val, __s, __radix);
    }
}
#endif

/**  \ingroup avr_stdlib
Highest number that can be generated by random(). */
#define RANDOM_MAX 0x7FFFFFFF

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

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

     If no seed value is provided, the functions are automatically seeded with
     a value of 1.
*/
extern long random(void);
/**
 \ingroup avr_stdlib
   Pseudo-random number generator seeding; see random().
*/
extern void srandom(unsigned long __seed);

/**
 \ingroup avr_stdlib
   Variant of random() that stores the context in the user-supplied
   variable located at \c ctx instead of a static library variable
   so the function becomes re-entrant.
*/
extern long random_r(unsigned long *__ctx);
#endif /* __ASSEMBLER */
/**@}*/

/**@{*/
/** \name Conversion functions for double arguments. */
/** \ingroup avr_stdlib
    Bit value that can be passed in \c flags to ftostre(),
    dtostre() and ldtostre(). */
#define DTOSTR_ALWAYS_SIGN 0x01        /* put '+' or ' ' for positives */
/** \ingroup avr_stdlib
    Bit value that can be passed in \c flags to ftostre(),
    dtostre() and ldtostre(). */
#define DTOSTR_PLUS_SIGN   0x02        /* put '+' rather than ' ' */
/** \ingroup avr_stdlib
    Bit value that can be passed in \c flags to ftostre(),
    dtostre() and ldtostre(). */
#define DTOSTR_UPPERCASE   0x04        /* put 'E' rather 'e' */

#ifndef __ASSEMBLER__

/**
   \ingroup avr_stdlib
   The ftostre() function converts the \c float value passed in \c val into
   an ASCII representation that will be stored under \c s.  The caller
   is responsible for providing sufficient storage in \c s.

   Conversion is done in the format
   <tt>&quot;[-]d.ddde&plusmn;dd&quot;</tt> where there is
```

```
   one digit before the decimal-point character and the number of
   digits after it is equal to the precision \c prec; if the precision
   is zero, no decimal-point character appears.  If \c flags has the
   #DTOSTR_UPPERCASE bit set, the letter \c 'E' (rather than \c 'e' ) will be
   used to introduce the exponent.  The exponent always contains two
   digits; if the value is zero, the exponent is \c "00".

   If \c flags has the #DTOSTR_ALWAYS_SIGN bit set, a space character
   will be placed into the leading position for positive numbers.

   If \c flags has the #DTOSTR_PLUS_SIGN bit set, a plus sign will be
   used instead of a space character in this case.

   The ftostre() function returns the pointer to the converted string \c s.
*/
extern char *ftostre(float __val, char *__s, unsigned char __prec,
                     unsigned char __flags);
/**
   \ingroup avr_stdlib
   The dtostre() function is similar to the ftostre() function, except that
   it converts a \c double value instead of a \c float value.

   dtostre() is currently only supported when \c double is a 32-bit type. */
extern char *dtostre(double __val, char *__s, unsigned char __prec,
             unsigned char __flags);
/**
   \ingroup avr_stdlib
   The ldtostre() function is similar to the ftostre() function, except that
   it converts a \c long \c double value instead of a \c float value.

   ldtostre() is currently only supported when \c long \c double is a
   32-bit type. */
extern char *ldtostre(long double __val, char *__s, unsigned char __prec,
             unsigned char __flags);

/**
   \ingroup avr_stdlib
   The ftostrf() function converts the \c float value passed in \c val into
   an ASCII representationthat will be stored in \c s.  The caller
   is responsible for providing sufficient storage in \c s.

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

   The ftostrf() function returns the pointer to the converted string \c s.
*/
extern char *ftostrf(float __val, signed char __width,
                     unsigned char __prec, char *__s);
/**
   \ingroup avr_stdlib
   The dtostrf() function is similar to the ftostrf() function, except that
   converts a \c double value instead of a \c float value.

   ldtostre() is currently only supported when \c double is a 32-bit type. */
extern char *dtostrf(double __val, signed char __width,
                     unsigned char __prec, char *__s);
/**
   \ingroup avr_stdlib
   The ldtostrf() function is similar to the ftostrf() function, except that
   converts a \c long \c double value instead of a \c float value.

   ldtostre() is currently only supported when \c long \c double is a
   32-bit type. */
extern char *ldtostrf(long double __val, signed char __width,
                      unsigned char __prec, char *__s);

/**
   \ingroup avr_stdlib
    Successful termination for exit(); evaluates to 0.
*/
#define EXIT_SUCCESS 0

/**
   \ingroup avr_stdlib
    Unsuccessful termination for exit(); evaluates to a non-zero value.
*/
#define EXIT_FAILURE 1

/**@}*/

#ifndef __DOXYGEN__
/* dummy declarations for libstdc++ compatibility */
extern int system (const char *);
extern char *getenv (const char *);
```

```
#endif  /* __DOXYGEN__ */

#ifdef __cplusplus
}
#endif

#endif /* __ASSEMBLER */

#endif /* _STDLIB_H_ */
```

23.58 string.h File Reference

Macros

- #define _FFS(x)

Functions

- int ffs (int __val)
- int ffsl (long __val)
- int ffsll (long long __val)
- void ∗ memccpy (void ∗, const void ∗, int, size_t)
- void ∗ memchr (const void ∗, int, size_t)
- int memcmp (const void ∗, const void ∗, size_t)
- void ∗ memcpy (void ∗, const void ∗, size_t)
- void ∗ memmem (const void ∗, size_t, const void ∗, size_t)
- void ∗ memmove (void ∗, const void ∗, size_t)
- void ∗ memrchr (const void ∗, int, size_t)
- void ∗ memset (void ∗, int, size_t)
- char ∗ strcat (char ∗, const char ∗)
- char ∗ strchr (const char ∗, int)
- char ∗ strchrnul (const char ∗, int)
- int strcmp (const char ∗, const char ∗)
- char ∗ strcpy (char ∗, const char ∗)
- int strcasecmp (const char ∗, const char ∗)
- char ∗ strcasestr (const char ∗, const char ∗)
- size_t strcspn (const char ∗__s, const char ∗__reject)
- char ∗ strdup (const char ∗s1)
- char ∗ strndup (const char ∗s, size_t n)
- size_t strlcat (char ∗, const char ∗, size_t)
- size_t strlcpy (char ∗, const char ∗, size_t)
- size_t strlen (const char ∗)
- char ∗ strlwr (char ∗)
- char ∗ strncat (char ∗, const char ∗, size_t)
- int strncmp (const char ∗, const char ∗, size_t)
- char ∗ strncpy (char ∗, const char ∗, size_t)
- int strncasecmp (const char ∗, const char ∗, size_t)
- size_t strnlen (const char ∗, size_t)
- char ∗ strpbrk (const char ∗__s, const char ∗__accept)
- char ∗ strrchr (const char ∗, int)
- char ∗ strrev (char ∗)
- char ∗ strsep (char ∗∗, const char ∗)
- size_t strspn (const char ∗__s, const char ∗__accept)
- char ∗ strstr (const char ∗, const char ∗)
- char ∗ strtok (char ∗, const char ∗)
- char ∗ strtok_r (char ∗, const char ∗, char ∗∗)
- char ∗ strupr (char ∗)

23.59 string.h

Go to the documentation of this file.

```
/* Copyright (c) 2002,2007 Marek Michalkiewicz
   All rights reserved.

   Redistribution and use in source and binary forms, with or without
   modification, are permitted provided that the following conditions are met:

   * Redistributions of source code must retain the above copyright
     notice, this list of conditions and the following disclaimer.

   * Redistributions in binary form must reproduce the above copyright
     notice, this list of conditions and the following disclaimer in
     the documentation and/or other materials provided with the
     distribution.

   * Neither the name of the copyright holders nor the names of
     contributors may be used to endorse or promote products derived
     from this software without specific prior written permission.

  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
  AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
  IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
  ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
  LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
  CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
  SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
  INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
  CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
  ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
  POSSIBILITY OF SUCH DAMAGE. */

/* $Id$ */

/*
   string.h

   Contributors:
     Created by Marek Michalkiewicz <marekm@linux.org.pl>
 */

#ifndef _STRING_H_
#define _STRING_H_ 1

#ifndef __DOXYGEN__
#define __need_NULL
#define __need_size_t
#include <stddef.h>

#ifndef __ATTR_PURE__
#define __ATTR_PURE__ __attribute__((__pure__))
#endif

#ifndef __ATTR_CONST__
# define __ATTR_CONST__ __attribute__((__const__))
#endif
#endif  /* !__DOXYGEN__ */

#ifdef __cplusplus
extern "C" {
#endif

/** \file */
/** \defgroup avr_string <string.h>: Strings
    \code #include <string.h> \endcode

    The string functions perform string operations on \c NULL terminated
    strings.

    \note If the strings you are working on resident in program space (flash),
    you will need to use the string functions described in \ref avr_pgmspace. */


/** \ingroup avr_string

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

    This macro is very similar to the function ffs() except that
    it evaluates its argument at compile-time, so it should only
    be applied to compile-time constant expressions where it will
    reduce to a constant itself.
    Application of this macro to expressions that are not constant
    at compile-time is not recommended, and might result in a huge
    amount of code generated.
```

```

    \returns The _FFS() macro returns the position of the first
    (least significant) bit set in the word val, or 0 if no bits are set.
    The least significant bit is position 1.  Only 16 bits of argument
    are evaluted.
*/
#if defined(__DOXYGEN__)
#define _FFS(x)
#else  /* !DOXYGEN */
#define _FFS(x) \
    (1                \
     + (((x) & 1) == 0)      \
     + (((x) & 3) == 0)      \
     + (((x) & 7) == 0)      \
     + (((x) & 017) == 0)       \
     + (((x) & 037) == 0)       \
     + (((x) & 077) == 0)       \
     + (((x) & 0177) == 0)      \
     + (((x) & 0377) == 0)      \
     + (((x) & 0777) == 0)      \
     + (((x) & 01777) == 0)     \
     + (((x) & 03777) == 0)     \
     + (((x) & 07777) == 0)     \
     + (((x) & 017777) == 0)    \
     + (((x) & 037777) == 0)    \
     + (((x) & 077777) == 0)    \
     - (((x) & 0177777) == 0) * 16)
#endif /* DOXYGEN */

/** \ingroup avr_string
    \fn int ffs(int val);

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

    \returns The ffs() function returns the position of the first
    (least significant) bit set in the word \p val, or 0 if no bits are set.
    The least significant bit is position 1.

    \note For expressions that are constant at compile time, consider
    using the \ref _FFS macro instead.
*/
extern int ffs(int __val) __ATTR_CONST__;

/** \ingroup avr_string
    \fn int ffsl(long val);

    \brief Same as ffs(), for an argument of type long. */
extern int ffsl(long __val) __ATTR_CONST__;

/** \ingroup avr_string
    \fn int ffsll(long long val);

    \brief Same as ffs(), for an argument of type <tt>long long</tt>. */
__extension__ extern int ffsll(long long __val) __ATTR_CONST__;

/** \ingroup avr_string
    \fn void *memccpy(void *dest, const void *src, int val, size_t len)
    \brief Copy memory area.

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

    \returns The memccpy() function returns a pointer to the next character
    in \p dest after \p val, or \c NULL if \p val was not found in the first
    \p len characters of \p src. */
extern void *memccpy(void *, const void *, int, size_t);

/** \ingroup avr_string
    \fn void *memchr(const void *src, int val, size_t len)
    \brief Scan memory for a character.

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

    \returns The memchr() function returns a pointer to the matching byte or
    NULL if the character does not occur in the given memory area.  */
extern void *memchr(const void *, int, size_t) __ATTR_PURE__;

/** \ingroup avr_string
    \fn int memcmp(const void *s1, const void *s2, size_t len)
    \brief Compare memory areas

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

```

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

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

    \warning This function is not -mint8 compatible, although if you only care
    about testing for equality, this function should be safe to use. */
extern int memcmp(const void *, const void *, size_t) __ATTR_PURE__;

/** \ingroup avr_string
    \fn void *memcpy(void *dest, const void *src, size_t len)
    \brief Copy a memory area.

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

    \returns The memcpy() function returns a pointer to dest.  */
extern void *memcpy(void *, const void *, size_t);

/** \ingroup avr_string
    \fn void *memmem(const void *s1, size_t len1, const void *s2, size_t len2)

    The memmem() function finds the start of the first occurrence of the
    substring \p s2 of length \p len2 in the memory area \p s1 of length
    \p len1.

    \return The memmem() function returns a pointer to the beginning of
    the substring, or \c NULL if the substring is not found. If \p len2
    is zero, the function returns \p s1. */
extern void *memmem(const void *, size_t, const void *, size_t) __ATTR_PURE__;

/** \ingroup avr_string
    \fn void *memmove(void *dest, const void *src, size_t len)
    \brief Copy memory area.

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

    \returns The memmove() function returns a pointer to dest.  */
extern void *memmove(void *, const void *, size_t);

/** \ingroup avr_string
    \fn void *memrchr(const void *src, int val, size_t len)

    The memrchr() function is like the memchr() function, except that it
    searches backwards from the end of the \p len bytes pointed to by \p
    src instead of forwards from the front. (Glibc, GNU extension.)

    \return The memrchr() function returns a pointer to the matching
    byte or \c NULL if the character does not occur in the given memory
    area. */
extern void *memrchr(const void *, int, size_t) __ATTR_PURE__;

/** \ingroup avr_string
    \fn void *memset(void *dest, int val, size_t len)
    \brief Fill memory with a constant byte.

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

    \returns The memset() function returns a pointer to the memory area dest. */
extern void *memset(void *, int, size_t);

/** \ingroup avr_string
    \fn char *strcat(char *dest, const char *src)
    \brief Concatenate two strings.

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

    \returns The strcat() function returns a pointer to the resulting string
    dest.  */
extern char *strcat(char *, const char *);

/** \ingroup avr_string
    \fn char *strchr(const char *src, int val)
    \brief Locate character in string.

    \returns The strchr() function returns a pointer to the first occurrence of
    the character \p val in the string \p src, or \c NULL if the character
    is not found.
    <br><br>
```

```
    Here "character" means "byte" -- these functions do not work with
    wide or multi-byte characters.  */
extern char *strchr(const char *, int) __ATTR_PURE__;

/** \ingroup avr_string
    \fn char *strchrnul(const char *s, int c)

    The strchrnul() function is like strchr() except that if \p c is not
    found in \p s, then it returns a pointer to the null byte at the end
    of \p s, rather than \c NULL. (Glibc, GNU extension.)

    \return The strchrnul() function returns a pointer to the matched
    character, or a pointer to the null byte at the end of \p s (i.e.,
    \c s+strlen(s)) if the character is not found.  */
extern char *strchrnul(const char *, int) __ATTR_PURE__;

/** \ingroup avr_string
    \fn int strcmp(const char *s1, const char *s2)
    \brief Compare two strings.

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

    \returns The strcmp() function returns an integer less than, equal
    to, or greater than zero if \p s1 is found, respectively, to be less
    than, to match, or be greater than \p s2. A consequence of the
    ordering used by strcmp() is that if \p s1 is an initial substring
    of \p s2, then \p s1 is considered to be "less than" \p s2. */
extern int strcmp(const char *, const char *) __ATTR_PURE__;

/** \ingroup avr_string
    \fn char *strcpy(char *dest, const char *src)
    \brief Copy a string.

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

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

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

/** \ingroup avr_string
    \fn int strcasecmp(const char *s1, const char *s2)
    \brief Compare two strings ignoring case.

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

    \returns The strcasecmp() function returns an integer less than,
    equal to, or greater than zero if \p s1 is found, respectively, to
    be less than, to match, or be greater than \p s2. A consequence of
    the ordering used by strcasecmp() is that if \p s1 is an initial
    substring of \p s2, then \p s1 is considered to be "less than"
    \p s2. */
extern int strcasecmp(const char *, const char *) __ATTR_PURE__;

/** \ingroup avr_string
    \fn char *strcasestr(const char *s1, const char *s2)

    The strcasestr() function finds the first occurrence of the
    substring \p s2 in the string \p s1. This is like strstr(), except
    that it ignores case of alphabetic symbols in searching for the
    substring. (Glibc, GNU extension.)

    \return The strcasestr() function returns a pointer to the beginning
    of the substring, or \c NULL if the substring is not found. If \p s2
    points to a string of zero length, the function returns \p s1. */
extern char *strcasestr(const char *, const char *) __ATTR_PURE__;

/** \ingroup avr_string
    \fn size_t strcspn(const char *s, const char *reject)

    The strcspn() function calculates the length of the initial segment
    of \p s which consists entirely of characters not in \p reject.

    \return The strcspn() function returns the number of characters in
    the initial segment of \p s which are not in the string \p reject.
    The terminating zero is not considered as a part of string. */
extern size_t strcspn(const char *__s, const char *__reject) __ATTR_PURE__;

/** \ingroup avr_string
```

```
    \fn char *strdup(const char *s1)
    \brief Duplicate a string.

    The strdup() function allocates memory and copies into it the string
    addressed by \p s1, including the terminating null character.

    \warning The strdup() function calls malloc() to allocate the memory
    for the duplicated string! The user is responsible for freeing the
    memory by calling free().

    \returns The strdup() function returns a pointer to the resulting string
    dest. If malloc() cannot allocate enough storage for the string, strdup()
    will return \c NULL.

    \warning Be sure to check the return value of the strdup() function to
    make sure that the function has succeeded in allocating the memory!
*/
extern char *strdup(const char *s1);

/** \ingroup avr_string
    \fn char *strndup(const char *s, size_t len)
    \brief Duplicate a string.

    The strndup() function is similar to strdup(), but copies at most
    \p len bytes. If \p s is longer than \p len, only \p len bytes are copied,
    and a terminating null byte (<tt>'\0'</tt>) is added.

    Memory for the new string is obtained with malloc(), and can be freed
    with free().

    \returns The strndup() function returns the location of the newly malloc'ed
    memory, or \c NULL if the allocation failed.
*/
extern char *strndup(const char *s, size_t n);

/** \ingroup avr_string
    \fn size_t strlcat(char *dst, const char *src, size_t siz)
    \brief Concatenate two strings.

    Appends \p src to string \p dst of size \p siz (unlike strncat(),
    \p siz is the full size of \p dst, not space left).  At most \p siz-1
    characters will be copied.  Always \p '\\0' terminated (unless \p siz <=
    \p strlen(dst)).

    \returns The strlcat() function returns strlen(src) + MIN(siz,
    strlen(initial dst)).  If retval >= siz, truncation occurred.  */
extern size_t strlcat(char *, const char *, size_t);

/** \ingroup avr_string
    \fn size_t strlcpy(char *dst, const char *src, size_t siz)
    \brief Copy a string.

    Copy \p src to string \p dst of size \p siz.  At most \p siz-1
    characters will be copied.  Always '\\0' terminated (unless \p siz == 0).

    \returns The strlcpy() function returns strlen(src). If retval >= siz,
    truncation occurred.  */
extern size_t strlcpy(char *, const char *, size_t);

/** \ingroup avr_string
    \fn size_t strlen(const char *src)
    \brief Calculate the length of a string.

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

    \returns The strlen() function returns the number of characters in
    \p src.  */
extern size_t strlen(const char *) __ATTR_PURE__;

/** \ingroup avr_string
    \fn char *strlwr(char *s)
    \brief Convert a string to lower case.

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

    \returns The strlwr() function returns a pointer to the converted
    string.  Conversion is perfomed in-place.  */
extern char *strlwr(char *);

/** \ingroup avr_string
    \fn char *strncat(char *dest, const char *src, size_t len)
    \brief Concatenate two strings.

    The strncat() function is similar to strcat(), except that only the first
```

```
    \p len characters of \p src are appended to \p dest.

    \returns The strncat() function returns a pointer to the resulting string
    \p dest.  */
extern char *strncat(char *, const char *, size_t);

/** \ingroup avr_string
    \fn int strncmp(const char *s1, const char *s2, size_t len)
    \brief Compare two strings.

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

    \returns The strncmp() function returns an integer less than, equal to, or
    greater than zero if \p s1 (or the first \p len bytes thereof) is found,
    respectively, to be less than, to match, or be greater than \p s2.  */
extern int strncmp(const char *, const char *, size_t) __ATTR_PURE__;

/** \ingroup avr_string
    \fn char *strncpy(char *dest, const char *src, size_t len)
    \brief Copy a string.

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

    In the case where the length of \p src is less than that of \p len,
    the remainder of \p dest will be padded with nulls (<tt>'\0'</tt>).

    \returns The strncpy() function returns a pointer to the destination
    string \p dest.  */
extern char *strncpy(char *, const char *, size_t);

/** \ingroup avr_string
    \fn int strncasecmp(const char *s1, const char *s2, size_t len)
    \brief Compare two strings ignoring case.

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

    \returns The strncasecmp() function returns an integer less than,
    equal to, or greater than zero if \p s1 (or the first \p len bytes
    thereof) is found, respectively, to be less than, to match, or be
    greater than \p s2. A consequence of the ordering used by
    strncasecmp() is that if \p s1 is an initial substring of \p s2,
    then \p s1 is considered to be "less than" \p s2.  */
extern int strncasecmp(const char *, const char *, size_t) __ATTR_PURE__;

/** \ingroup avr_string
    \fn size_t strnlen(const char *src, size_t len)
    \brief Determine the length of a fixed-size string.

    The strnlen() function returns the number of characters in the string
    pointed to by \p src, not including the terminating '\\0' character, but at
    most \p len. In doing this, strnlen() looks only at the first \p len
    characters at \p src and never beyond \p src + \p len.

    \returns The strnlen function returns strlen(src), if that is less than
    \p len, or \p len if there is no '\\0' character among the first \p len
    characters pointed to by \p src. */
extern size_t strnlen(const char *, size_t) __ATTR_PURE__;

/** \ingroup avr_string
    \fn char *strpbrk(const char *s, const char *accept)

    The strpbrk() function locates the first occurrence in the string
    \p s of any of the characters in the string \p accept.

    \return  The strpbrk() function returns a pointer to the character
    in \p s that matches one of the characters in \p accept, or \c NULL
    if no such character is found. The terminating zero is not
    considered as a part of string: if one or both args are empty, the
    result will be \c NULL. */
extern char *strpbrk(const char *__s, const char *__accept) __ATTR_PURE__;

/** \ingroup avr_string
    \fn char *strrchr(const char *src, int val)
    \brief Locate character in string.

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

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

    \returns The strrchr() function returns a pointer to the matched character
    or \c NULL if the character is not found. */
```

```
extern char *strrchr(const char *, int) __ATTR_PURE__;

/** \ingroup avr_string
    \fn char *strrev(char *s)
    \brief Reverse a string.

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

    \returns The strrev() function returns a pointer to the beginning of the
    reversed string.  */
extern char *strrev(char *);

/** \ingroup avr_string
    \fn char *strsep(char **sp, const char *delim)
    \brief Parse a string into tokens.

    The strsep() function locates, in the string referenced by \p *sp,
    the first occurrence of any character in the string \p delim (or the
    terminating ’\\0’ character) and replaces it with a ’\\0’.  The
    location of the next character after the delimiter character (or \c
    NULL, if the end of the string was reached) is stored in \p *sp. An
    “empty” field, i.e. one caused by two adjacent delimiter
    characters, can be detected by comparing the location referenced by
    the pointer returned in \p *sp to ’\\0’.

    \return The strsep() function returns a pointer to the original
    value of \p *sp. If \p *sp is initially \c NULL, strsep() returns
    \c NULL. */
extern char *strsep(char **, const char *);

/** \ingroup avr_string
    \fn size_t strspn(const char *s, const char *accept)

    The strspn() function calculates the length of the initial segment
    of \p s which consists entirely of characters in \p accept.

    \return  The strspn() function returns the number of characters in
    the initial segment of \p s which consist only of characters from \p
    accept. The terminating zero is not considered as a part of string. */
extern size_t strspn(const char *__s, const char *__accept) __ATTR_PURE__;

/** \ingroup avr_string
    \fn char *strstr(const char *s1, const char *s2)
    \brief Locate a substring.

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

    \returns The strstr() function returns a pointer to the beginning of
    the substring, or \c NULL if the substring is not found. If \p s2
    points to a string of zero length, the function returns \p s1. */
extern char *strstr(const char *, const char *) __ATTR_PURE__;

/** \ingroup avr_string
    \fn char *strtok(char *s, const char *delim)
    \brief Parses the string s into tokens.

    strtok parses the string s into tokens. The first call to strtok
    should have s as its first argument. Subsequent calls should have
    the first argument set to \c NULL. If a token ends with a delimiter, this
    delimiting character is overwritten with a ’\\0’ and a pointer to the next
    character is saved for the next call to strtok. The delimiter string
    delim may be different for each call.

    \returns The strtok() function returns a pointer to the next token or
    \c NULL when no more tokens are found.

    \note strtok() is NOT reentrant. For a reentrant version of this function
    see \c strtok_r().
*/
extern char *strtok(char *, const char *);

/** \ingroup avr_string
    \fn char *strtok_r(char *string, const char *delim, char **last)
    \brief Parses string into tokens.

    strtok_r parses string into tokens. The first call to strtok_r
    should have string as its first argument. Subsequent calls should have
    the first argument set to \c NULL. If a token ends with a delimiter, this
    delimiting character is overwritten with a ’\\0’ and a pointer to the next
    character is saved for the next call to strtok_r. The delimiter string
    \p delim may be different for each call. \p last is a user allocated char*
    pointer. It must be the same while parsing the same string. strtok_r is
    a reentrant version of strtok().

    \returns The strtok_r() function returns a pointer to the next token or
```

```
    \c NULL when no more tokens are found. */
extern char *strtok_r(char *, const char *, char **);

/** \ingroup avr_string
    \fn char *strupr(char *s)
    \brief Convert a string to upper case.

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

    \returns The strupr() function returns a pointer to the converted
    string.  The pointer is the same as that passed in since the operation is
    perform in place. */
extern char *strupr(char *);

#ifndef __DOXYGEN__
/* libstdc++ compatibility, dummy declarations */
extern int strcoll(const char *s1, const char *s2);
extern char *strerror(int errnum);
extern size_t strxfrm(char *dest, const char *src, size_t n);
#endif  /* !__DOXYGEN__ */

#ifdef __cplusplus
}
#endif

#endif /* _STRING_H_ */

```

23.60 time.h File Reference

Data Structures

- struct tm
- struct week_date

Macros

- #define ONE_HOUR 3600
- #define ONE_DEGREE 3600
- #define ONE_DAY 86400
- #define UNIX_OFFSET 946684800
- #define NTP_OFFSET 3155673600

Typedefs

- typedef uint32_t time_t

Enumerations

- enum _WEEK_DAYS_ {
  SUNDAY , **MONDAY** , **TUESDAY** , **WEDNESDAY** ,
  THURSDAY , **FRIDAY** , **SATURDAY** }
- enum _MONTHS_ {
  JANUARY , **FEBRUARY** , **MARCH** , **APRIL** ,
  MAY , **JUNE** , **JULY** , **AUGUST** ,
  SEPTEMBER , **OCTOBER** , **NOVEMBER** , **DECEMBER** }

Functions

- time_t time (time_t *timer)
- int32_t difftime (time_t time1, time_t time0)
- time_t mktime (struct tm *timeptr)
- time_t mk_gmtime (const struct tm *timeptr)
- struct tm * gmtime (const time_t *timer)
- void gmtime_r (const time_t *timer, struct tm *timeptr)
- struct tm * localtime (const time_t *timer)
- void localtime_r (const time_t *timer, struct tm *timeptr)
- char * asctime (const struct tm *timeptr)
- void asctime_r (const struct tm *timeptr, char *buf)
- char * ctime (const time_t *timer)
- void ctime_r (const time_t *timer, char *buf)
- char * isotime (const struct tm *tmptr)
- void isotime_r (const struct tm *, char *)
- size_t strftime (char *s, size_t maxsize, const char *format, const struct tm *timeptr)
- void set_dst (int(*)(const time_t *, int32_t *))
- void set_zone (int32_t)
- void set_system_time (time_t timestamp)
- void system_tick (void)
- uint8_t is_leap_year (int16_t year)
- uint8_t month_length (int16_t year, uint8_t month)
- uint8_t week_of_year (const struct tm *timeptr, uint8_t start)
- uint8_t week_of_month (const struct tm *timeptr, uint8_t start)
- struct week_date * iso_week_date (int year, int yday)
- void iso_week_date_r (int year, int yday, struct week_date *)
- uint32_t fatfs_time (const struct tm *timeptr)
- void set_position (int32_t latitude, int32_t longitude)
- int16_t equation_of_time (const time_t *timer)
- int32_t daylight_seconds (const time_t *timer)
- time_t solar_noon (const time_t *timer)
- time_t sun_rise (const time_t *timer)
- time_t sun_set (const time_t *timer)
- float solar_declinationf (const time_t *timer)
- double solar_declination (const time_t *timer)
- long double solar_declinationl (const time_t *timer)
- int8_t moon_phase (const time_t *timer)
- unsigned long gm_sidereal (const time_t *timer)
- unsigned long lm_sidereal (const time_t *timer)

23.61 time.h

Go to the documentation of this file.

```
/*
 * (C)2012 Michael Duane Rice All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions are
 * met:
 *
 * Redistributions of source code must retain the above copyright notice, this
 * list of conditions and the following disclaimer. Redistributions in binary
 * form must reproduce the above copyright notice, this list of conditions
 * and the following disclaimer in the documentation and/or other materials
 * provided with the distribution. Neither the name of the copyright holders
 * nor the names of contributors may be used to endorse or promote products
 * derived from this software without specific prior written permission.
 *
```

```c
 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
 * AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
 * LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
 * CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
 * SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
 * INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
 * CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
 * ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
 * POSSIBILITY OF SUCH DAMAGE.
 */

/* $Id$ */

#ifndef TIME_H
#define TIME_H

#ifdef __cplusplus
extern  "C" {
#endif

#include <stdint.h>
#include <stdlib.h>

/** \file */

/** \defgroup avr_time <time.h>: Time
    \code #include <time.h> \endcode
    <h3>Introduction to the Time functions</h3>
    This file declares the time functions implemented in AVR-LibC.

    The implementation aspires to conform with ISO/IEC 9899 (C90). However, due to limitations of the
    target processor and the nature of its development environment, a practical implementation must
    of necessity deviate from the standard.

    Section 7.23.2.1 clock()
    The type clock_t, the macro CLOCKS_PER_SEC, and the function clock() are not implemented. We
    consider these items belong to operating system code, or to application code when no operating
    system is present.

    Section 7.23.2.3 mktime()
    The standard specifies that mktime() should return (time_t) -1, if the time cannot be represented.
    This implementation always returns a 'best effort' representation.

    Section 7.23.2.4 time()
    The standard specifies that time() should return (time_t) -1, if the time is not available.
    Since the application must initialize the time system, this functionality is not implemented.

    Section 7.23.2.2, difftime()
    Due to the lack of a 64 bit double, the function difftime() returns a long integer. In most cases
    this change will be invisible to the user, handled automatically by the compiler.

    Section 7.23.1.4 struct tm
    Per the standard, struct tm->tm_isdst is greater than zero when Daylight Saving time is in effect.
    This implementation further specifies that, when positive, the value of tm_isdst represents
    the amount time is advanced during Daylight Saving time.

    Section 7.23.3.5 strftime()
    Only the 'C' locale is supported, therefore the modifiers 'E' and 'O' are ignored.
    The 'Z' conversion is also ignored, due to the lack of time zone name.

    In addition to the above departures from the standard, there are some behaviors which are different
    from what is often expected, though allowed under the standard.

    There is no 'platform standard' method to obtain the current time, time zone, or
    daylight savings 'rules' in the AVR environment. Therefore the application must initialize
    the time system with this information. The functions set_zone(), set_dst(), and
    set_system_time() are provided for initialization. Once initialized, system time is maintained by
    calling the function system_tick() at one second intervals.

    Though not specified in the standard, it is often expected that time_t is a signed integer
    representing an offset in seconds from Midnight Jan 1 1970... i.e. 'Unix time'. This implementation
    uses an unsigned 32 bit integer offset from Midnight Jan 1 2000. The use of this 'epoch' helps to
    simplify the conversion functions, while the 32 bit value allows time to be properly represented
    until Tue Feb 7 06:28:15 2136 UTC. The macros UNIX_OFFSET and NTP_OFFSET are defined to assist in
    converting to and from Unix and NTP time stamps.

    Unlike desktop counterparts, it is impractical to implement or maintain the 'zoneinfo' database.
    Therefore no attempt is made to account for time zone, daylight saving, or leap seconds in past dates.
    All calculations are made according to the currently configured time zone and daylight saving 'rule'.

    In addition to C standard functions, re-entrant versions of ctime(), asctime(), gmtime() and
```

```
    localtime() are provided which, in addition to being re-entrant, have the property of claiming
    less permanent storage in RAM. An additional time conversion, isotime() and its re-entrant version,
    uses far less storage than either ctime() or asctime().

    Along with the usual smattering of utility functions, such as is_leap_year(), this library includes
    a set of functions related the sun and moon, as well as sidereal time functions.
*/

/** \ingroup avr_time */
/**@{*/

/**
    time_t represents seconds elapsed from Midnight, Jan 1 2000 UTC (the Y2K 'epoch').
    Its range allows this implementation to represent time up to Tue Feb 7 06:28:15 2136 UTC.
*/
typedef uint32_t time_t;

/**
    The time function returns the systems current time stamp.
    If timer is not a null pointer, the return value is also assigned to the object it points to.
*/
time_t time(time_t *timer);

/**
    The difftime function returns the difference between two binary time stamps,
    time1 - time0.
*/
int32_t difftime(time_t time1, time_t time0);


/**
    The tm structure contains a representation of time 'broken down' into components of the
    Gregorian calendar.

    The value of tm_isdst is zero if Daylight Saving Time is not in effect, and is negative if
    the information is not available.

    When Daylight Saving Time is in effect, the value represents the number of
    seconds the clock is advanced.

    See the set_dst() function for more information about Daylight Saving.
*/
struct tm {
    int8_t          tm_sec; /**< seconds after the minute - [ 0 to 59 ] */
    int8_t          tm_min; /**< minutes after the hour - [ 0 to 59 ] */
    int8_t          tm_hour; /**< hours since midnight - [ 0 to 23 ] */
    int8_t          tm_mday; /**< day of the month - [ 1 to 31 ] */
    int8_t          tm_wday; /**< days since Sunday - [ 0 to 6 ] */
    int8_t          tm_mon; /**< months since January - [ 0 to 11 ] */
    int16_t         tm_year; /**< years since 1900 */
    int16_t         tm_yday; /**< days since January 1 - [ 0 to 365 ] */
    int16_t         tm_isdst; /**< Daylight Saving Time flag */
};

#ifndef __DOXYGEN__
/* We have to provide clock_t / CLOCKS_PER_SEC so that libstdc++-v3 can
   be built.  We define CLOCKS_PER_SEC via a symbol _CLOCKS_PER_SEC_
   so that the user can provide the value on the link line, which should
   result in little or no run-time overhead compared with a constant.  */
typedef unsigned long clock_t;
extern char *_CLOCKS_PER_SEC_;
#define CLOCKS_PER_SEC ((clock_t) _CLOCKS_PER_SEC_)
extern clock_t clock(void);
#endif  /* !__DOXYGEN__ */

/**
    This function 'compiles' the elements of a broken-down time structure, returning a binary time stamp.
    The elements of timeptr are interpreted as representing Local Time.

    The original values of the tm_wday and tm_yday elements of the structure are ignored,
    and the original values of the other elements are not restricted to the ranges stated for struct tm.

    The element tm_isdst is used for input and output. If set to 0 or a positive value on input, this
    requests calculation for Daylight Savings Time being off or on, respectively. If set to a negative
    value on input, it requests calculation to return whether Daylight Savings Time is in effect or
    not according to the other values.

    On successful completion, the values of all elements of timeptr are set to the appropriate range.
*/
time_t          mktime(struct tm * timeptr);

/**
    This function 'compiles' the elements of a broken-down time structure, returning a binary time
```

```
    stamp.
    The elements of timeptr are interpreted as representing UTC.

    The original values of the tm_wday and tm_yday elements of the structure are ignored,
    and the original values of the other elements are not restricted to the ranges stated for struct
    tm.

    Unlike mktime(), this function DOES NOT modify the elements of timeptr.
*/
time_t          mk_gmtime(const struct tm * timeptr);

/**
    The gmtime function converts the time stamp pointed to by timer into broken-down time,
    expressed as UTC.
*/
struct tm      *gmtime(const time_t * timer);

/**
    Re entrant version of gmtime().
*/
void            gmtime_r(const time_t * timer, struct tm * timeptr);

/**
    The localtime function converts the time stamp pointed to by timer into broken-down time,
    expressed as Local time.
*/
struct tm      *localtime(const time_t * timer);

/**
    Re entrant version of localtime().
*/
void            localtime_r(const time_t * timer, struct tm * timeptr);

/**
    The asctime function converts the broken-down time of timeptr, into an ascii string in the form

    Sun Mar 23 01:03:52 2013
*/
char           *asctime(const struct tm * timeptr);

/**
    Re entrant version of asctime().
*/
void            asctime_r(const struct tm * timeptr, char *buf);

/**
    The ctime function is equivalent to asctime(localtime(timer))
*/
char           *ctime(const time_t * timer);

/**
    Re entrant version of ctime().
*/
void            ctime_r(const time_t * timer, char *buf);

/**
The isotime function constructs an ascii string in the form
    \code2013-03-23 01:03:52\endcode
*/
char           *isotime(const struct tm * tmptr);

/**
    Re entrant version of isotime()
*/
void            isotime_r(const struct tm *, char *);

/**
    A complete description of strftime() is beyond the pale of this document.
    Refer to ISO/IEC document 9899 for details.

    All conversions are made using the 'C Locale', ignoring the E or O modifiers. Due to the lack of
    a time zone 'name', the 'Z' conversion is also ignored.
*/
size_t          strftime(char *s, size_t maxsize, const char *format, const struct tm * timeptr);

/**
    Specify the Daylight Saving function.

    The Daylight Saving function should examine its parameters to determine whether
    Daylight Saving is in effect, and return a value appropriate for tm_isdst.

    Working examples for the USA and the EU are available..

        \code #include <util/eu_dst.h>\endcode
        for the European Union, and
        \code #include <util/usa_dst.h>\endcode
        for the United States
```

```

    If a Daylight Saving function is not specified, the system will ignore Daylight Saving.
*/
void            set_dst(int (*) (const time_t *, int32_t *));

/**
    Set the 'time zone'. The parameter is given in seconds East of the Prime Meridian.
    Example for New York City:
    \code set_zone(-5 * ONE_HOUR);\endcode

    If the time zone is not set, the time system will operate in UTC only.
*/
void            set_zone(int32_t);

/**
    Initialize the system time. Examples are...

    From a Clock / Calendar type RTC:
    \code
    struct tm rtc_time;

    read_rtc(&rtc_time);
    rtc_time.tm_isdst = 0;
    set_system_time( mktime(&rtc_time) );
    \endcode

    From a Network Time Protocol time stamp:
    \code
    set_system_time(ntp_timestamp - NTP_OFFSET);
    \endcode

    From a UNIX time stamp:
     \code
    set_system_time(unix_timestamp - UNIX_OFFSET);
    \endcode

*/
void            set_system_time(time_t timestamp);

/**
    Maintain the system time by calling this function at a rate of 1 Hertz.

    It is anticipated that this function will typically be called from within an
    Interrupt Service Routine, (though that is not required). It therefore includes code which
    makes it simple to use from within a 'Naked' ISR, avoiding the cost of saving and restoring
    all the cpu registers.

    Such an ISR may resemble the following example...
    \code
        ISR(RTC_OVF_vect, ISR_NAKED)
        {
            system_tick();
            reti();
        }
    \endcode
*/
void            system_tick(void);

/**
    Enumerated labels for the days of the week.
*/
enum _WEEK_DAYS_ {
    SUNDAY,
    MONDAY,
    TUESDAY,
    WEDNESDAY,
    THURSDAY,
    FRIDAY,
    SATURDAY
};

/**
    Enumerated labels for the months.
*/
enum _MONTHS_ {
    JANUARY,
    FEBRUARY,
    MARCH,
    APRIL,
    MAY,
    JUNE,
    JULY,
    AUGUST,
    SEPTEMBER,
    OCTOBER,
    NOVEMBER,
    DECEMBER
```

```
};

/**
    Return 1 if year is a leap year, zero if it is not.
*/
uint8_t         is_leap_year(int16_t year);

/**
    Return the length of month, given the year and month, where month is in the range 1 to 12.
*/
uint8_t         month_length(int16_t year, uint8_t month);

/**
    Return the calendar week of year, where week 1 is considered to begin on the
    day of week specified by 'start'. The returned value may range from zero to 52.
*/
uint8_t         week_of_year(const struct tm * timeptr, uint8_t start);

/**
    Return the calendar week of month, where the first week is considered to begin on the
    day of week specified by 'start'. The returned value may range from zero to 5.
*/
uint8_t         week_of_month(const struct tm * timeptr, uint8_t start);

/**
    Structure which represents a date as a year, week number of that year, and day of week.
    See http://en.wikipedia.org/wiki/ISO_week_date for more information.
*/
struct week_date {
    int year; /**< year number (Gregorian calendar) */
    int week; /**< week number (#1 is where first Thursday is in) */
    int day; /**< day within week */
};

/**
    Return a week_date structure with the ISO_8601 week based date corresponding to the given
    year and day of year. See http://en.wikipedia.org/wiki/ISO_week_date for more
    information.
*/
struct week_date * iso_week_date( int year, int yday);

/**
    Re-entrant version of iso-week_date.
*/
void iso_week_date_r( int year, int yday, struct week_date *);

/**
    Convert a Y2K time stamp into a FAT file system time stamp.
*/
uint32_t        fatfs_time(const struct tm * timeptr);

/** One hour, expressed in seconds */
#define ONE_HOUR 3600

/** Angular degree, expressed in arc seconds */
#define ONE_DEGREE 3600

/** One day, expressed in seconds */
#define ONE_DAY 86400

/** Difference between the Y2K and the UNIX epochs, in seconds. To convert a Y2K
    timestamp to UNIX...
    \code
    long unix;
    time_t y2k;

    y2k = time(NULL);
    unix = y2k + UNIX_OFFSET;
    \endcode
*/
#define UNIX_OFFSET 946684800

/** Difference between the Y2K and the NTP epochs, in seconds. To convert a Y2K
    timestamp to NTP...
    \code
    unsigned long ntp;
    time_t y2k;

    y2k = time(NULL);
    ntp = y2k + NTP_OFFSET;
    \endcode
*/
#define NTP_OFFSET 3155673600

/*
 * ====================================================================
 *                              Ephemera
```

```
 */

/**
    Set the geographic coordinates of the 'observer', for use with several of the
    following functions. Parameters are passed as seconds of North Latitude, and seconds
    of East Longitude.

    For New York City...
    \code set_position( 40.7142 * ONE_DEGREE, -74.0064 * ONE_DEGREE); \endcode
*/
void set_position(int32_t latitude, int32_t longitude);

/**
    Computes the difference between apparent solar time and mean solar time.
    The returned value is in seconds.
*/
int16_t equation_of_time(const time_t * timer);

/**
    Computes the amount of time the sun is above the horizon, at the location of the observer.

    NOTE: At observer locations inside a polar circle, this value can be zero during the winter,
    and can exceed ONE_DAY during the summer.

    The returned value is in seconds.
*/
int32_t daylight_seconds(const time_t * timer);

/**
    Computes the time of solar noon, at the location of the observer.
*/
time_t solar_noon(const time_t * timer);

/**
    Return the time of sunrise, at the location of the observer. See the note about
daylight_seconds().
*/
time_t sun_rise(const time_t * timer);

/**
    Return the time of sunset, at the location of the observer. See the note about daylight_seconds().
*/
time_t sun_set(const time_t * timer);

/**
    Returns the declination of the sun in radians.
*/
float           solar_declinationf(const time_t * timer);

/**
    Returns the declination of the sun in radians.

    This implementation is only available when \c double is a 32-bit type.
*/
double          solar_declination(const time_t * timer);

/**
    Returns the declination of the sun in radians.

    This implementation is only available when <tt>long double</tt> is
    a 32-bit type.
*/
long double     solar_declinationl(const time_t * timer);

/**
    Returns an approximation to the phase of the moon.
    The sign of the returned value indicates a waning or waxing phase.
    The magnitude of the returned value indicates the percentage illumination.
*/
int8_t moon_phase(const time_t * timer);

/**
    Returns Greenwich Mean Sidereal Time, as seconds into the sidereal day.
    The returned value will range from 0 through 86399 seconds.
*/
unsigned long gm_sidereal(const time_t * timer);

/**
    Returns Local Mean Sidereal Time, as seconds into the sidereal day.
    The returned value will range from 0 through 86399 seconds.
*/
unsigned long   lm_sidereal(const time_t * timer);

/**@}*/
#ifdef __cplusplus
}
#endif
```

```
#endif /* TIME_H  */
```

23.62 atomic.h File Reference

Macros

- #define ATOMIC_BLOCK(type)
- #define NONATOMIC_BLOCK(type)
- #define ATOMIC_RESTORESTATE
- #define ATOMIC_FORCEON
- #define NONATOMIC_RESTORESTATE
- #define NONATOMIC_FORCEOFF

23.63 atomic.h

Go to the documentation of this file.

```
/* Copyright (c) 2007 Dean Camera
   All rights reserved.

   Redistribution and use in source and binary forms, with or without
   modification, are permitted provided that the following conditions are met:

     * Redistributions of source code must retain the above copyright
       notice, this list of conditions and the following disclaimer.

     * Redistributions in binary form must reproduce the above copyright
       notice, this list of conditions and the following disclaimer in
       the documentation and/or other materials provided with the
       distribution.

     * Neither the name of the copyright holders nor the names of
       contributors may be used to endorse or promote products derived
       from this software without specific prior written permission.

  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
  AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
  IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
  ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
  LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
  CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
  SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
  INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
  CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
  ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
  POSSIBILITY OF SUCH DAMAGE.
*/

/* $Id$ */

#ifndef _UTIL_ATOMIC_H_
#define _UTIL_ATOMIC_H_ 1

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

#if !defined(__DOXYGEN__)
/* Internal helper functions. */
static __inline__ uint8_t __iSeiRetVal(void)
{
    sei();
    return 1;
}

static __inline__ uint8_t __iCliRetVal(void)
{
    cli();
    return 1;
}

static __inline__ void __iSeiParam(const uint8_t *__s)
{
    sei();
```

```
    __asm__ volatile ("" ::: "memory");
    (void)__s;
}

static __inline__ void __iCliParam(const uint8_t *__s)
{
    cli();
    __asm__ volatile ("" ::: "memory");
    (void)__s;
}

static __inline__ void __iRestore(const  uint8_t *__s)
{
    SREG = *__s;
    __asm__ volatile ("" ::: "memory");
}
#endif  /* !__DOXYGEN__ */

/** \file */
/** \defgroup util_atomic <util/atomic.h> Atomically and Non-Atomically Executed Code Blocks

    \code
    #include <util/atomic.h>
    \endcode

    \note The macros in this header file require the ISO/IEC 9899:1999
    ("ISO C99") feature of for loop variables that are declared inside
    the for loop itself.  For that reason, this header file can only
    be used if the standard level of the compiler (option --std=) is
    set to either \c c99, \c gnu99 or higher.

    The macros in this header file deal with code blocks that are
    guaranteed to be executed Atomically or Non-Atomically.  The term
    "Atomic" in this context refers to the inability of the respective
    code to be interrupted.

    These macros operate via automatic manipulation of the Global
    Interrupt Status (I) bit of the SREG register. Exit paths from
    both block types are all managed automatically without the need
    for special considerations, i.e. the interrupt status will be
    restored to the same value it had when entering the respective
    block (unless ATOMIC_FORCEON or NONATOMIC_FORCEOFF are used).
    \warning The features in this header are implemented by means of
    a for loop.  This means that commands like \c break and \c continue
    that are located in an atomic block refer to the atomic for loop,
    not to a loop construct that hosts the atomic block.

    A typical example that requires atomic access is a 16 (or more)
    bit variable that is shared between the main execution path and an
    ISR.  While declaring such a variable as volatile ensures that the
    compiler will not optimize accesses to it away, it does not
    guarantee atomic access to it.  Assuming the following example:

    \code
#include <stdint.h>
#include <avr/interrupt.h>
#include <avr/io.h>

volatile uint16_t ctr;

ISR(TIMER1_OVF_vect)
{
  ctr--;
}

...
int
main(void)
{
   ...
   ctr = 0x200;
   start_timer();
   while (ctr != 0)
     // wait
       ;
   ...
}
    \endcode

    There is a chance where the main context will exit its wait loop
    when the variable \c ctr just reached the value 0xFF.  This happens
    because the compiler cannot natively access a 16-bit variable
    atomically in an 8-bit CPU.  So the variable is for example at
    0x100, the compiler then tests the low byte for 0, which succeeds.
    It then proceeds to test the high byte, but that moment the ISR
    triggers, and the main context is interrupted.  The ISR will
    decrement the variable from 0x100 to 0xFF, and the main context
```

```
    proceeds.  It now tests the high byte of the variable which is
    (now) also 0, so it concludes the variable has reached 0, and
    terminates the loop.

    Using the macros from this header file, the above code can be
    rewritten like:

    \code
#include <stdint.h>
#include <avr/interrupt.h>
#include <avr/io.h>
#include <util/atomic.h>

volatile uint16_t ctr;

ISR(TIMER1_OVF_vect)
{
  ctr--;
}

...
int
main(void)
{
   ...
   ctr = 0x200;
   start_timer();
   sei();
   uint16_t ctr_copy;
   do
   {
     ATOMIC_BLOCK(ATOMIC_FORCEON)
     {
       ctr_copy = ctr;
     }
   }
   while (ctr_copy != 0);
   ...
}
    \endcode

    This will install the appropriate interrupt protection before
    accessing variable \c ctr, so it is guaranteed to be consistently
    tested.  If the global interrupt state were uncertain before
    entering the #ATOMIC_BLOCK, it should be executed with the
    parameter #ATOMIC_RESTORESTATE rather than #ATOMIC_FORCEON.

    See \ref optim_code_reorder for things to be taken into account
    with respect to compiler optimizations.
*/

/** \def ATOMIC_BLOCK(type)
    \ingroup util_atomic

    Creates a block of code that is guaranteed to be executed
    atomically. Upon entering the block the Global Interrupt Status
    flag in SREG is disabled, and re-enabled upon exiting the block
    from any exit path.

    Two possible macro parameters are permitted, #ATOMIC_RESTORESTATE
    and #ATOMIC_FORCEON.
*/
#if defined(__DOXYGEN__)
#define ATOMIC_BLOCK(type)
#else
#define ATOMIC_BLOCK(type) for ( type, __ToDo = __iCliRetVal(); \
                           __ToDo ; __ToDo = 0 )
#endif  /* __DOXYGEN__ */

/** \def NONATOMIC_BLOCK(type)
    \ingroup util_atomic

    Creates a block of code that is executed non-atomically. Upon
    entering the block the Global Interrupt Status flag in SREG is
    enabled, and disabled upon exiting the block from any exit
    path. This is useful when nested inside ATOMIC_BLOCK sections,
    allowing for non-atomic execution of small blocks of code while
    maintaining the atomic access of the other sections of the parent
    ATOMIC_BLOCK.

    Two possible macro parameters are permitted,
    #NONATOMIC_RESTORESTATE and #NONATOMIC_FORCEOFF.
*/
#if defined(__DOXYGEN__)
#define NONATOMIC_BLOCK(type)
#else
#define NONATOMIC_BLOCK(type) for ( type, __ToDo = __iSeiRetVal(); \
```

```
                               __ToDo ;  __ToDo = 0 )
#endif  /* __DOXYGEN__ */

/** \def ATOMIC_RESTORESTATE
    \ingroup util_atomic

    This is a possible parameter for #ATOMIC_BLOCK. When used, it will
    cause the ATOMIC_BLOCK to restore the previous state of the SREG
    register, saved before the Global Interrupt Status flag bit was
    disabled. The net effect of this is to make the ATOMIC_BLOCK's
    contents guaranteed atomic, without changing the state of the
    Global Interrupt Status flag when execution of the block
    completes.
*/
#if defined(__DOXYGEN__)
#define ATOMIC_RESTORESTATE
#else
#define ATOMIC_RESTORESTATE uint8_t sreg_save \
    __attribute__((__cleanup__(__iRestore))) = SREG
#endif  /* __DOXYGEN__ */

/** \def ATOMIC_FORCEON
    \ingroup util_atomic

    This is a possible parameter for #ATOMIC_BLOCK. When used, it will
    cause the ATOMIC_BLOCK to force the state of the SREG register on
    exit, enabling the Global Interrupt Status flag bit. This saves a
    small amount of flash space, a register, and one or more processor
    cycles, since the previous value of the SREG register does not need
    to be saved at the start of the block.

    Care should be taken that ATOMIC_FORCEON is only used when it is
    known that interrupts are enabled before the block's execution or
    when the side effects of enabling global interrupts at the block's
    completion are known and understood.
*/
#if defined(__DOXYGEN__)
#define ATOMIC_FORCEON
#else
#define ATOMIC_FORCEON uint8_t sreg_save \
    __attribute__((__cleanup__(__iSeiParam))) = 0
#endif  /* __DOXYGEN__ */

/** \def NONATOMIC_RESTORESTATE
    \ingroup util_atomic

    This is a possible parameter for #NONATOMIC_BLOCK. When used, it
    will cause the NONATOMIC_BLOCK to restore the previous state of
    the SREG register, saved before the Global Interrupt Status flag
    bit was enabled. The net effect of this is to make the
    NONATOMIC_BLOCK's contents guaranteed non-atomic, without changing
    the state of the Global Interrupt Status flag when execution of
    the block completes.
*/
#if defined(__DOXYGEN__)
#define NONATOMIC_RESTORESTATE
#else
#define NONATOMIC_RESTORESTATE uint8_t sreg_save \
    __attribute__((__cleanup__(__iRestore))) = SREG
#endif  /* __DOXYGEN__ */

/** \def NONATOMIC_FORCEOFF
    \ingroup util_atomic

    This is a possible parameter for #NONATOMIC_BLOCK. When used, it
    will cause the NONATOMIC_BLOCK to force the state of the SREG
    register on exit, disabling the Global Interrupt Status flag
    bit.  This saves a small amout of flash space, a register, and one
    or more processor cycles, since the previous value of the SREG
    register does not need to be saved at the start of the block.

    Care should be taken that NONATOMIC_FORCEOFF is only used when it
    is known that interrupts are disabled before the block's execution
    or when the side effects of disabling global interrupts at the
    block's completion are known and understood.
*/
#if defined(__DOXYGEN__)
#define NONATOMIC_FORCEOFF
#else
#define NONATOMIC_FORCEOFF uint8_t sreg_save \
    __attribute__((__cleanup__(__iCliParam))) = 0
#endif  /* __DOXYGEN__ */

#endif
```

23.64 crc16.h File Reference

Functions

- static uint16_t _crc16_update (uint16_t __crc, uint8_t __data)
- static uint16_t _crc_xmodem_update (uint16_t __crc, uint8_t __data)
- static uint16_t _crc_ccitt_update (uint16_t __crc, uint8_t __data)
- static uint8_t _crc_ibutton_update (uint8_t __crc, uint8_t __data)
- static uint8_t _crc8_ccitt_update (uint8_t __crc, uint8_t __data)

23.65 crc16.h

Go to the documentation of this file.

```
/* Copyright (c) 2002, 2003, 2004  Marek Michalkiewicz
   Copyright (c) 2005, 2007 Joerg Wunsch
   Copyright (c) 2013 Dave Hylands
   Copyright (c) 2013 Frederic Nadeau
   All rights reserved.

   Redistribution and use in source and binary forms, with or without
   modification, are permitted provided that the following conditions are met:

   * Redistributions of source code must retain the above copyright
     notice, this list of conditions and the following disclaimer.

   * Redistributions in binary form must reproduce the above copyright
     notice, this list of conditions and the following disclaimer in
     the documentation and/or other materials provided with the
     distribution.

   * Neither the name of the copyright holders nor the names of
     contributors may be used to endorse or promote products derived
     from this software without specific prior written permission.

  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
  AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
  IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
  ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
  LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
  CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
  SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
  INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
  CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
  ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
  POSSIBILITY OF SUCH DAMAGE. */

/* $Id$ */

#ifndef _UTIL_CRC16_H_
#define _UTIL_CRC16_H_

#include <stdint.h>

#ifndef __DOXYGEN__
#ifndef __ATTR_ALWAYS_INLINE__
#define __ATTR_ALWAYS_INLINE__ __inline__ __attribute__((__always_inline__))
#endif
#endif /* ! DOXYGEN */

/** \file */
/** \defgroup util_crc <util/crc16.h>: CRC Computations
    \code#include <util/crc16.h>\endcode

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

    \par References:

    \par

    See the Dallas Semiconductor app note 27 for 8051 assembler example and
    general CRC optimization suggestions. The table on the last page of the
    app note is the key to understanding these implementations.

    \par

    Jack Crenshaw's "Implementing CRCs" article in the January 1992 isue of \e
```

```
    Embedded \e Systems \e Programming. This may be difficult to find, but it
    explains CRC's in very clear and concise terms. Well worth the effort to
    obtain a copy.

    A typical application would look like:

    \code
    // Dallas iButton test vector.
    uint8_t serno[] = { 0x02, 0x1c, 0xb8, 0x01, 0, 0, 0, 0xa2 };

    int
    checkcrc (void)
    {
        uint8_t crc = 0, i;

        for (i = 0; i < sizeof serno / sizeof serno[0]; i++)
            crc = _crc_ibutton_update (crc, serno[i]);

        return crc; // must be 0
    }
    \endcode
*/

/** \ingroup util_crc
    Optimized CRC-16 calculation.

    Polynomial: x<sup>16</sup> + x<sup>15</sup> + x<sup>2</sup> + 1 (0xa001)<br>
    Initial value: \c 0xffff

    This CRC is normally used in disk-drive controllers.

    The following is the equivalent functionality written in C.

    \code
    uint16_t
    crc16_update (uint16_t crc, uint8_t a)
    {
        crc ^= a;
        for (int i = 0; i < 8; ++i)
        {
            if (crc & 1)
                crc = (crc » 1) ^ 0xA001;
            else
                crc = crc » 1;
        }

        return crc;
    }
    \endcode */

static __ATTR_ALWAYS_INLINE__ uint16_t
_crc16_update(uint16_t __crc, uint8_t __data)
{
    uint8_t __tmp;
    uint16_t __ret;

    __asm__ __volatile__ (
        "eor %A0,%2" "\n\t"
        "mov %1,%A0" "\n\t"
        "swap %1" "\n\t"
        "eor %1,%A0" "\n\t"
        "mov __tmp_reg__,%1" "\n\t"
        "lsr %1" "\n\t"
        "lsr %1" "\n\t"
        "eor %1,__tmp_reg__" "\n\t"
        "mov __tmp_reg__,%1" "\n\t"
        "lsr %1" "\n\t"
        "eor %1,__tmp_reg__" "\n\t"
        "andi %1,0x07" "\n\t"
        "mov __tmp_reg__,%A0" "\n\t"
        "mov %A0,%B0" "\n\t"
        "lsr %1" "\n\t"
        "ror __tmp_reg__" "\n\t"
        "ror %1" "\n\t"
        "mov %B0,__tmp_reg__" "\n\t"
        "eor %A0,%1" "\n\t"
        "lsr __tmp_reg__" "\n\t"
        "ror %1" "\n\t"
        "eor %B0,__tmp_reg__" "\n\t"
        "eor %A0,%1"
        : "=r" (__ret), "=d" (__tmp)
        : "r" (__data), "0" (__crc)
        : "r0"
    );
    return __ret;
}

```

```
/** \ingroup util_crc
    Optimized CRC-XMODEM calculation.

    Polynomial: x<sup>16</sup> + x<sup>12</sup> + x<sup>5</sup> + 1 (0x1021)<br>
    Initial value: \c 0x0

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

    The following is the equivalent functionality written in C.

    \code
    uint16_t
    crc_xmodem_update (uint16_t crc, uint8_t data)
    {
        crc = crc ^ ((uint16_t)data « 8);
        for (int i = 0; i < 8; i++)
        {
            if (crc & 0x8000)
                crc = (crc « 1) ^ 0x1021;
            else
                crc «= 1;
        }

        return crc;
    }
    \endcode */
static __ATTR_ALWAYS_INLINE__ uint16_t
_crc_xmodem_update (uint16_t __crc, uint8_t __data)
{
    uint16_t __ret;             /* %B0:%A0 (alias for __crc) */
    uint8_t __tmp1;             /* %1 */
    uint8_t __tmp2;             /* %2 */
                                /* %3  __data */

    __asm__ __volatile__ (
        "eor    %B0,%3"          "\n\t"
        "mov    %1,%A0"          "\n\t"
        "mov    %2,%B0"          "\n\t"

        "mov    %A0,%B0"         "\n\t"
        "swap   %B0"             "\n\t"
        "eor    %A0,%B0"         "\n\t"

        "andi   %A0,0xf0"        "\n\t"
        "andi   %B0,0x0f"        "\n\t"

        "eor    %1,%A0"          "\n\t"
        "eor    %2,%B0"          "\n\t"

        "lsl    %A0"             "\n\t"
        "rol    %B0"             "\n\t"

        "eor    %B0,%1"          "\n\t"
        "eor    %A0,%2"
        : "=d" (__ret), "=r" (__tmp1), "=r" (__tmp2)
        : "r" (__data), "0" (__crc)
    );
    return __ret;
}

/** \ingroup util_crc
    Optimized CRC-CCITT calculation.

    Polynomial: x<sup>16</sup> + x<sup>12</sup> + x<sup>5</sup> + 1 (0x8408)<br>
    Initial value: \c 0xffff

    This is the CRC used by PPP and IrDA.

    See RFC1171 (PPP protocol) and IrDA IrLAP 1.1

    \note Although the CCITT polynomial is the same as that used by the Xmodem
    protocol, they are quite different. The difference is in how the bits are
    shifted through the alorgithm. Xmodem shifts the MSB of the CRC and the
    input first, while CCITT shifts the LSB of the CRC and the input first.

    The following is the equivalent functionality written in C.

    \code
    uint16_t
    crc_ccitt_update (uint16_t crc, uint8_t data)
    {
        data ^= lo8 (crc);
        data ^= data « 4;

        return ((((uint16_t)data « 8) | hi8 (crc)) ^ (uint8_t)(data » 4)
                ^ ((uint16_t)data « 3));
```

```
    }
    \endcode */

static __ATTR_ALWAYS_INLINE__ uint16_t
_crc_ccitt_update (uint16_t __crc, uint8_t __data)
{
    uint16_t __ret;

    __asm__ __volatile__ (
        "eor    %A0,%1"          "\n\t"

        "mov    __tmp_reg__,%A0" "\n\t"
        "swap   %A0"             "\n\t"
        "andi   %A0,0xf0"        "\n\t"
        "eor    %A0,__tmp_reg__" "\n\t"

        "mov    __tmp_reg__,%B0" "\n\t"

        "mov    %B0,%A0"         "\n\t"

        "swap   %A0"             "\n\t"
        "andi   %A0,0x0f"        "\n\t"
        "eor    __tmp_reg__,%A0" "\n\t"

        "lsr    %A0"             "\n\t"
        "eor    %B0,%A0"         "\n\t"

        "eor    %A0,%B0"         "\n\t"
        "lsl    %A0"             "\n\t"
        "lsl    %A0"             "\n\t"
        "lsl    %A0"             "\n\t"
        "eor    %A0,__tmp_reg__"

        : "=d" (__ret)
        : "r" (__data), "0" (__crc)
        : "r0"
    );
    return __ret;
}

/** \ingroup util_crc
    Optimized Dallas (now Maxim) iButton 8-bit CRC calculation.

    Polynomial: x<sup>8</sup> + x<sup>5</sup> + x<sup>4</sup> + 1 (0x8C)<br>
    Initial value: \c 0x0

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

    The following is the equivalent functionality written in C.

    \code
    uint8_t
    _crc_ibutton_update (uint8_t crc, uint8_t data)
    {
        crc = crc ^ data;
        for (uint8_t i = 0; i < 8; i++)
        {
            if (crc & 0x01)
                crc = (crc » 1) ^ 0x8C;
            else
                crc »= 1;
        }

        return crc;
    }
    \endcode
*/

static __ATTR_ALWAYS_INLINE__ uint8_t
_crc_ibutton_update (uint8_t __crc, uint8_t __data)
{
    uint8_t __i, __pattern;
    __asm__ __volatile__ (
        "eor    %0, %4"     "\n\t"
        "ldi    %1, 8"      "\n\t"
        "ldi    %2, 0x8C"   "\n"
        "1: lsr %0"         "\n\t"
        "brcc   2f"         "\n\t"
        "eor    %0, %2"     "\n"
        "2: dec %1"         "\n\t"
        "brne   1b"
        : "=r" (__crc), "=d" (__i), "=d" (__pattern)
        : "0" (__crc), "r" (__data));
    return __crc;
}

/** \ingroup util_crc
```

```
    Optimized CRC-8-CCITT calculation.

    Polynomial: x<sup>8</sup> + x<sup>2</sup> + x + 1 (0xE0)<br>

    For use with simple CRC-8<br>
    Initial value: 0x0

    For use with CRC-8-ROHC<br>
    Initial value: 0xff<br>
    Reference: http://tools.ietf.org/html/rfc3095#section-5.9.1

    For use with CRC-8-ATM/ITU<br>
    Initial value: 0xff<br>
    Final XOR value: 0x55<br>
    Reference: http://www.itu.int/rec/T-REC-I.432.1-199902-I/en

    The C equivalent has been originally written by Dave Hylands.
    Assembly code is based on _crc_ibutton_update optimization.

    The following is the equivalent functionality written in C.

    \code
    uint8_t
    _crc8_ccitt_update (uint8_t inCrc, uint8_t inData)
    {
        uint8_t data = inCrc ^ inData;

        for (int i = 0; i < 8; i++)
        {
            if ((data & 0x80) != 0)
            {
                data «= 1;
                data ^= 0x07;
            }
            else
            {
                data «= 1;
            }
        }
        return data;
    }
    \endcode
*/

static __ATTR_ALWAYS_INLINE__ uint8_t
_crc8_ccitt_update(uint8_t __crc, uint8_t __data)
{
    uint8_t __i, __pattern;
    __asm__ __volatile__ (
        "eor    %0, %4"    "\n\t"
        "ldi    %1, 8"     "\n\t"
        "ldi    %2, 0x07"  "\n"
        "1: lsl    %0"     "\n\t"
        "brcc   2f"        "\n\t"
        "eor    %0, %2"    "\n"
        "2: dec    %1"     "\n\t"
        "brne   1b"
        : "=r" (__crc), "=d" (__i), "=d" (__pattern)
        : "0" (__crc), "r" (__data));
    return __crc;
}

#endif /* _UTIL_CRC16_H_ */
```

23.66 delay.h File Reference

Macros

- #define F_CPU 1000000UL

Functions

- static void _delay_ms (double __ms)
- static void _delay_us (double __us)

23.67 delay.h

Go to the documentation of this file.

```
/* Copyright (c) 2002, Marek Michalkiewicz
   Copyright (c) 2004,2005,2007 Joerg Wunsch
   Copyright (c) 2007  Florin-Viorel Petrov
   All rights reserved.

   Redistribution and use in source and binary forms, with or without
   modification, are permitted provided that the following conditions are met:

   * Redistributions of source code must retain the above copyright
     notice, this list of conditions and the following disclaimer.

   * Redistributions in binary form must reproduce the above copyright
     notice, this list of conditions and the following disclaimer in
     the documentation and/or other materials provided with the
     distribution.

   * Neither the name of the copyright holders nor the names of
     contributors may be used to endorse or promote products derived
     from this software without specific prior written permission.

  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
  AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
  IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
  ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
  LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
  CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
  SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
  INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
  CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
  ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
  POSSIBILITY OF SUCH DAMAGE. */

/* $Id$ */

#ifndef _UTIL_DELAY_H_
#define _UTIL_DELAY_H_ 1

#ifndef __DOXYGEN__
#  ifndef __HAS_DELAY_CYCLES
#    define __HAS_DELAY_CYCLES 1
#  endif

#  ifndef __ATTR_ALWAYS_INLINE__
#    define __ATTR_ALWAYS_INLINE__ __inline__ __attribute__((__always_inline__))
#  endif

#endif  /* __DOXYGEN__ */

#include <stdint.h>
#include <util/delay_basic.h>

/** \file */
/** \defgroup util_delay <util/delay.h>: Convenience functions for busy-wait delay loops
    \code
    #define F_CPU 1000000UL  // 1 MHz
    //#define F_CPU 14.7456e6
    #include <util/delay.h>
    \endcode

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

    The functions in this header file are wrappers around the basic
    busy-wait functions from <tt>\<util/delay_basic.h\></tt>.  They are meant as
    convenience functions where actual time values can be specified
    rather than a number of cycles to wait for.  The idea behind is
    that compile-time constant expressions will be eliminated by
    compiler optimization so floating-point expressions can be used
    to calculate the number of delay cycles needed based on the CPU
    frequency passed by the macro F_CPU.

    \note In order for these functions to work as intended, compiler
    optimizations <em>must</em> be enabled, and the delay time
    <em>must</em> be an expression that is a known constant at
    compile-time.  If these requirements are not met, the resulting
    delay will be much longer (and basically unpredictable), and
    applications that otherwise do not use floating-point calculations
    will experience severe code bloat by the floating-point library
    routines linked into the application.

    The functions available allow the specification of microsecond, and
```

```
    millisecond delays directly, using the application-supplied macro
    F_CPU as the CPU clock frequency (in Hertz).
*/


#ifndef F_CPU
/* prevent compiler error by supplying a default */
# warning "F_CPU not defined for <util/delay.h>"
/** \ingroup util_delay
    \def F_CPU
    \brief CPU frequency in Hz

    The macro F_CPU specifies the CPU frequency to be considered by
    the delay macros.  This macro is normally supplied by the
    environment (e.g. from within a project header, or the project's
    Makefile).  The value 1 MHz here is only provided as a "vanilla"
    fallback if no such user-provided definition could be found.

    In terms of the delay functions, the CPU frequency can be given as
    a floating-point constant (e.g. 3.6864e6 for 3.6864 MHz).
    However, the macros in <util/setbaud.h> require it to be an
    integer value.
 */
# define F_CPU 1000000UL
#endif

#ifndef __OPTIMIZE__
# warning "Compiler optimizations disabled; functions from <util/delay.h> won't work as designed"
#endif

/**
   \ingroup util_delay

   Perform a delay of \c __ms milliseconds, using _delay_loop_2().

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

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

   When the user request delay which exceed the maximum possible one,
   _delay_ms() provides a decreased resolution functionality. In this
   mode _delay_ms() will work with a resolution of 1/10 ms, providing
   delays up to 6.5535 seconds (independent from CPU frequency).  The
   user will not be informed about decreased resolution.

   If the avr-gcc toolchain has \c __builtin_avr_delay_cycles()
   support, maximal possible delay is 4294967.295 ms/ F_CPU in MHz. For
   values greater than the maximal possible delay, overflow may result in
   no delay i.e., 0 ms.

   Conversion of \c __ms into clock cycles may not always result in
   an integral value.  By default, the clock cycles are rounded up to the next
   integer. This ensures that the user gets at least \c __ms
   microseconds of delay.

   Alternatively, by defining the macro \c __DELAY_ROUND_DOWN__, or
   \c __DELAY_ROUND_CLOSEST__, before including this header file, the
   algorithm can be made to round down, or round to closest integer,
   respectively.

   \note The implementation of _delay_ms() based on
   \c __builtin_avr_delay_cycles() is not backward compatible with older
   implementations.  In order to get functionality backward compatible
   with previous versions, the macro \c __DELAY_BACKWARD_COMPATIBLE__
   must be defined before including this header file.
 */
static __ATTR_ALWAYS_INLINE__ void _delay_ms(double __ms);

void
_delay_ms(double __ms)
{
    double __tmp ;
#if __HAS_DELAY_CYCLES && defined(__OPTIMIZE__) \
  && !defined(__DELAY_BACKWARD_COMPATIBLE__)
    uint32_t __ticks_dc;
    extern void __builtin_avr_delay_cycles(uint32_t);
    __tmp = ((F_CPU) / 1e3) * __ms;

    #if defined(__DELAY_ROUND_DOWN__)
        __ticks_dc = (uint32_t)__builtin_fabs(__tmp);

    #elif defined(__DELAY_ROUND_CLOSEST__)
        __ticks_dc = (uint32_t)(__builtin_fabs(__tmp)+0.5);

    #else
        //round up by default
```

```
        __ticks_dc = (uint32_t)(__builtin_ceil(__builtin_fabs(__tmp)));
    #endif

    __builtin_avr_delay_cycles(__ticks_dc);

#else
    uint16_t __ticks;
    __tmp = ((F_CPU) / 4e3) * __ms;
    if (__tmp < 1.0)
        __ticks = 1;
    else if (__tmp > 65535)
    {
        //  __ticks = requested delay in 1/10 ms
        __ticks = (uint16_t) (__ms * 10.0);
        while(__ticks)
        {
            // wait 1/10 ms
            _delay_loop_2(((F_CPU) / 4e3) / 10);
            __ticks --;
        }
        return;
    }
    else
        __ticks = (uint16_t)__tmp;
    _delay_loop_2(__ticks);
#endif
}

/**
   \ingroup util_delay

   Perform a delay of \c __us microseconds, using _delay_loop_1().

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

   The maximal possible delay is 768 &mu;s / F_CPU in MHz.

   If the user requests a delay greater than the maximal possible one,
   _delay_us() will automatically call _delay_ms() instead.  The user
   will not be informed about this case.

   If the avr-gcc toolchain has __builtin_avr_delay_cycles()
   support, maximal possible delay is 4294967.295 &mu;s/ F_CPU in MHz. For
   values greater than the maximal possible delay, overflow may result in
   no delay i.e., 0 &mu;s.

   Conversion of \c __us into clock cycles may not always result in
   integer.  By default, the clock cycles are rounded up to next
   integer. This ensures that the user gets at least \c __us
   microseconds of delay.

   Alternatively, by defining the macro \c __DELAY_ROUND_DOWN__, or
   \c __DELAY_ROUND_CLOSEST__, before including this header file, the
   algorithm can be made to round down, or round to closest integer,
   respectively.

   \note The implementation of _delay_us() based on
   \c __builtin_avr_delay_cycles() is not backward compatible with older
   implementations.  In order to get functionality backward compatible
   with previous versions, the macro \c __DELAY_BACKWARD_COMPATIBLE__
   must be defined before including this header file.
 */
static __ATTR_ALWAYS_INLINE__ void _delay_us(double __us);

void
_delay_us(double __us)
{
    double __tmp ;
#if __HAS_DELAY_CYCLES && defined(__OPTIMIZE__) \
  && !defined(__DELAY_BACKWARD_COMPATIBLE__)
    uint32_t __ticks_dc;
    extern void __builtin_avr_delay_cycles(uint32_t);
    __tmp = ((F_CPU) / 1e6) * __us;

    #if defined(__DELAY_ROUND_DOWN__)
        __ticks_dc = (uint32_t)__builtin_fabs(__tmp);

    #elif defined(__DELAY_ROUND_CLOSEST__)
        __ticks_dc = (uint32_t)(__builtin_fabs(__tmp)+0.5);

    #else
        //round up by default
        __ticks_dc = (uint32_t)(__builtin_ceil(__builtin_fabs(__tmp)));
    #endif

    __builtin_avr_delay_cycles(__ticks_dc);
```

```
#else
    uint8_t __ticks;
    double __tmp2 ;
    __tmp = ((F_CPU) / 3e6) * __us;
    __tmp2 = ((F_CPU) / 4e6) * __us;
    if (__tmp < 1.0)
        __ticks = 1;
    else if (__tmp2 > 65535)
    {
        _delay_ms(__us / 1000.0);
        return;
    }
    else if (__tmp > 255)
    {
        uint16_t __ticks=(uint16_t)__tmp2;
        _delay_loop_2(__ticks);
        return;
    }
    else
        __ticks = (uint8_t)__tmp;
    _delay_loop_1(__ticks);
#endif
}


#endif /* _UTIL_DELAY_H_ */
```

23.68 delay_basic.h File Reference

Functions

- void _delay_loop_1 (uint8_t __count)
- void _delay_loop_2 (uint16_t __count)

23.69 delay_basic.h

Go to the documentation of this file.

```
/* Copyright (c) 2002, Marek Michalkiewicz
   Copyright (c) 2007 Joerg Wunsch
   All rights reserved.

   Redistribution and use in source and binary forms, with or without
   modification, are permitted provided that the following conditions are met:

   * Redistributions of source code must retain the above copyright
     notice, this list of conditions and the following disclaimer.

   * Redistributions in binary form must reproduce the above copyright
     notice, this list of conditions and the following disclaimer in
     the documentation and/or other materials provided with the
     distribution.

   * Neither the name of the copyright holders nor the names of
     contributors may be used to endorse or promote products derived
     from this software without specific prior written permission.

  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
  AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
  IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
  ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
  LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
  CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
  SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
  INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
  CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
  ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
  POSSIBILITY OF SUCH DAMAGE. */

/* $Id$ */

#ifndef _UTIL_DELAY_BASIC_H_
#define _UTIL_DELAY_BASIC_H_ 1

#include <inttypes.h>

```

```c
#if !defined(__DOXYGEN__)
static __inline__ void _delay_loop_1(uint8_t __count) __attribute__((__always_inline__));
static __inline__ void _delay_loop_2(uint16_t __count) __attribute__((__always_inline__));
#endif

/** \file */
/** \defgroup util_delay_basic <util/delay_basic.h>: Basic busy-wait delay loops
    \code
    #include <util/delay_basic.h>
    \endcode

    The functions in this header file implement simple delay loops
    that perform a busy-waiting.  They are typically used to
    facilitate short delays in the program execution.  They are
    implemented as count-down loops with a well-known CPU cycle
    count per loop iteration.  As such, no other processing can
    occur simultaneously.  It should be kept in mind that the
    functions described here do not disable interrupts.

    In general, for long delays, the use of hardware timers is
    much preferrable, as they free the CPU, and allow for
    concurrent processing of other events while the timer is
    running.  However, in particular for very short delays, the
    overhead of setting up a hardware timer is too much compared
    to the overall delay time.

    Two inline functions are provided for the actual delay algorithms.

*/

/** \ingroup util_delay_basic

    Delay loop using an 8-bit counter \c __count, so up to 256
    iterations are possible.  (The value 256 would have to be passed
    as 0.)  The loop executes three CPU cycles per iteration, not
    including the overhead the compiler needs to setup the counter
    register.

    Thus, at a CPU speed of 1 MHz, delays of up to 768 microseconds
    can be achieved.
*/
void
_delay_loop_1(uint8_t __count)
{
    __asm__ volatile (
        "1: dec %0" "\n\t"
        "brne 1b"
        : "=r" (__count)
        : "0" (__count)
    );
}

/** \ingroup util_delay_basic

    Delay loop using a 16-bit counter \c __count, so up to 65536
    iterations are possible.  (The value 65536 would have to be
    passed as 0.)  The loop executes four CPU cycles per iteration,
    not including the overhead the compiler requires to setup the
    counter register pair.

    Thus, at a CPU speed of 1 MHz, delays of up to about 262.1
    milliseconds can be achieved.
 */
void
_delay_loop_2(uint16_t __count)
{
#if defined (__AVR_TINY__)
    __asm__ volatile (
        "1: subi %A0,1" "\n\t"
        "   sbci %B0,0" "\n\t"
        "brne 1b"
        : "+d" (__count)
    );
#else
    __asm__ volatile (
        "1: sbiw %0,1" "\n\t"
        "brne 1b"
        : "+w" (__count)
    );
#endif /* AVR_TINY */
}

#endif /* _UTIL_DELAY_BASIC_H_ */
```

23.70 eu_dst.h File Reference

Functions

- int eu_dst (const time_t *timer, int32_t *z)

23.71 eu_dst.h

Go to the documentation of this file.

```
/*
 * (c)2012 Michael Duane Rice All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions are
 * met:
 *
 * Redistributions of source code must retain the above copyright notice, this
 * list of conditions and the following disclaimer. Redistributions in binary
 * form must reproduce the above copyright notice, this list of conditions
 * and the following disclaimer in the documentation and/or other materials
 * provided with the distribution. Neither the name of the copyright holders
 * nor the names of contributors may be used to endorse or promote products
 * derived from this software without specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
 * AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
 * LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
 * CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
 * SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
 * INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
 * CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
 * ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
 * POSSIBILITY OF SUCH DAMAGE.
 */

/* $Id$ */

#ifndef EU_DST_H
#define EU_DST_H

#ifdef __cplusplus
extern "C" {
#endif

#include <time.h>
#include <stdint.h>

/** \file */
/** \defgroup eu_dst <util/eu_dst.h>: Daylight Saving function for the European Union.

    \code #include <util/eu_dst.h> \endcode
    Daylight Saving Time for the European Union */

/** \ingroup eu_dst
    \fn int eu_dst (const time_t *timer, int32_t *z)
    To utilize this function, call \code set_dst(eu_dst); \endcode

    Given the time stamp and time zone parameters provided, the Daylight
    Saving function must return a value appropriate for the tm structures'
    tm_isdst element. That is:

    - \c 0 : If Daylight Saving is not in effect.

    - \c -1 : If it cannot be determined if Daylight Saving is in effect.

    - A positive integer: Represents the number of seconds a clock is advanced
    for Daylight Saving.  This will typically be ONE_HOUR.

    Daylight Saving 'rules' are subject to frequent change.  For production
    applications it is recommended to write your own DST function, which
    uses 'rules' obtained from, and modifiable by, the end user (perhaps
    stored in EEPROM).
*/
int eu_dst (const time_t *timer, int32_t *z);

#ifdef __cplusplus
}
#endif

#endif
```

23.72 parity.h File Reference

Functions

- static uint8_t parity_even_bit (uint8_t __val)

23.73 parity.h

Go to the documentation of this file.

```
/* Copyright (c) 2002, Marek Michalkiewicz
   Copyright (c) 2004,2005,2007 Joerg Wunsch
   All rights reserved.

   Redistribution and use in source and binary forms, with or without
   modification, are permitted provided that the following conditions are met:

   * Redistributions of source code must retain the above copyright
     notice, this list of conditions and the following disclaimer.

   * Redistributions in binary form must reproduce the above copyright
     notice, this list of conditions and the following disclaimer in
     the documentation and/or other materials provided with the
     distribution.

   * Neither the name of the copyright holders nor the names of
     contributors may be used to endorse or promote products derived
     from this software without specific prior written permission.

  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
  AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
  IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
  ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
  LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
  CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
  SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
  INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
  CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
  ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
  POSSIBILITY OF SUCH DAMAGE. */

/* $Id$ */

#ifndef _UTIL_PARITY_H_
#define _UTIL_PARITY_H_

#include <stdint.h>

#ifndef __DOXYGEN__
#ifndef __ATTR_ALWAYS_INLINE__
#define __ATTR_ALWAYS_INLINE__ __inline__ __attribute__((__always_inline__))
#endif
#endif /* !DOXYGEN */

/** \file */
/** \defgroup util_parity <util/parity.h>: Parity bit generation
    \code #include <util/parity.h> \endcode

    This header file contains optimized assembler code to calculate
    the parity bit for a byte.
*/
/** \fn uint8_t parity_even_bit (uint8_t val);
    \ingroup util_parity
    \returns 1 if \c val has an odd number of bits set, and 0 otherwise. */

static __ATTR_ALWAYS_INLINE__
uint8_t parity_even_bit (uint8_t __val)
{
  if (__builtin_constant_p (__builtin_parity (__val)))
    return (uint8_t) __builtin_parity (__val);

  __asm (/* parity is in [0..7] */
         "mov  __tmp_reg__, %0"  "\n\t"
         "swap __tmp_reg__"      "\n\t"
         "eor  %0, __tmp_reg__"  "\n\t"
         /* parity is in [0..3] */
         "subi %0, -4"           "\n\t"
         "andi %0, -5"           "\n\t"
         "subi %0, -6"           "\n\t"
         /* parity is in [0,3] */
```

```
        "sbrc %0, 3"           "\n\t"
        "inc  %0"
        /* parity is in [0] */
        : "+d" (__val) :: "r0");

  return __val & 1;
}

#endif /* _UTIL_PARITY_H_ */
```

23.74 setbaud.h File Reference

Macros

- #define BAUD_TOL 2
- #define UBRR_VALUE
- #define UBRRL_VALUE
- #define UBRRH_VALUE
- #define USE_2X 0

23.75 setbaud.h

Go to the documentation of this file.

```
/* Copyright (c) 2007  Cliff Lawson
   Copyright (c) 2007  Carlos Lamas
   All rights reserved.

   Redistribution and use in source and binary forms, with or without
   modification, are permitted provided that the following conditions are met:

   * Redistributions of source code must retain the above copyright
     notice, this list of conditions and the following disclaimer.

   * Redistributions in binary form must reproduce the above copyright
     notice, this list of conditions and the following disclaimer in
     the documentation and/or other materials provided with the
     distribution.

   * Neither the name of the copyright holders nor the names of
     contributors may be used to endorse or promote products derived
     from this software without specific prior written permission.

  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
  AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
  IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
  ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
  LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
  CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
  SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
  INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
  CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
  ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
  POSSIBILITY OF SUCH DAMAGE. */

/* $Id$ */

/**
   \file
*/

/**
   \defgroup util_setbaud <util/setbaud.h>: Helper macros for baud rate calculations
   \code
   #define F_CPU 11059200
   #define BAUD 38400
   #include <util/setbaud.h>
   \endcode

   This header file requires that on entry values are already defined
   for F_CPU and BAUD.  In addition, the macro BAUD_TOL will define
   the baud rate tolerance (in percent) that is acceptable during
   the calculations.  The value of BAUD_TOL will default to 2 %.

   This header file defines macros suitable to setup the UART baud
```

```
   rate prescaler registers of an AVR.  All calculations are done
   using the C preprocessor.  Including this header file causes no
   other side effects so it is possible to include this file more than
   once (supposedly, with different values for the BAUD parameter),
   possibly even within the same function.

   Assuming that the requested BAUD is valid for the given F_CPU then
   the macro UBRR_VALUE is set to the required prescaler value.  Two
   additional macros are provided for the low and high bytes of the
   prescaler, respectively: UBRRL_VALUE is set to the lower byte of
   the UBRR_VALUE and UBRRH_VALUE is set to the upper byte.  An
   additional macro USE_2X will be defined.  Its value is set to 1 if
   the desired BAUD rate within the given tolerance could only be
   achieved by setting the U2X bit in the UART configuration.  It will
   be defined to 0 if U2X is not needed.

   Example usage:

   \code
   #include <avr/io.h>

   #define F_CPU 4000000

   static void
   uart_9600(void)
   {
   #define BAUD 9600
   #include <util/setbaud.h>
   UBRRH = UBRRH_VALUE;
   UBRRL = UBRRL_VALUE;
   #if USE_2X
   UCSRA |= (1 « U2X);
   #else
   UCSRA &= ~(1 « U2X);
   #endif
   }

   static void
   uart_38400(void)
   {
   #undef BAUD  // avoid compiler warning
   #define BAUD 38400
   #include <util/setbaud.h>
   UBRRH = UBRRH_VALUE;
   UBRRL = UBRRL_VALUE;
   #if USE_2X
   UCSRA |= (1 « U2X);
   #else
   UCSRA &= ~(1 « U2X);
   #endif
   }
   \endcode

   In this example, two functions are defined to setup the UART
   to run at 9600 Bd, and 38400 Bd, respectively.  Using a CPU
   clock of 4 MHz, 9600 Bd can be achieved with an acceptable
   tolerance without setting U2X (prescaler 25), while 38400 Bd
   require U2X to be set (prescaler 12).
*/

#ifndef F_CPU
#  error "setbaud.h requires F_CPU to be defined"
#endif

#ifndef BAUD
#  error "setbaud.h requires BAUD to be defined"
#endif

#if !(F_CPU)
#  error "F_CPU must be a constant value"
#endif

#if !(BAUD)
#  error "BAUD must be a constant value"
#endif

#if defined(__DOXYGEN__)
/**
   \def BAUD_TOL
   \ingroup util_setbaud

   Input and output macro for <util/setbaud.h>

   Define the acceptable baud rate tolerance in percent.  If not set
   on entry, it will be set to its default value of 2.
*/
#define BAUD_TOL 2
```

```

/**
   \def UBRR_VALUE
   \ingroup util_setbaud

   Output macro from <util/setbaud.h>

   Contains the calculated baud rate prescaler value for the UBRR
   register.
*/
#define UBRR_VALUE

/**
   \def UBRRL_VALUE
   \ingroup util_setbaud

   Output macro from <util/setbaud.h>

   Contains the lower byte of the calculated prescaler value
   (UBRR_VALUE).
*/
#define UBRRL_VALUE

/**
   \def UBRRH_VALUE
   \ingroup util_setbaud

   Output macro from <util/setbaud.h>

   Contains the upper byte of the calculated prescaler value
   (UBRR_VALUE).
*/
#define UBRRH_VALUE

/**
   \def USE_2X
   \ingroup util_setbaud

   Output macro from <util/setbaud.h>

   Contains the value 1 if the desired baud rate tolerance could only
   be achieved by setting the U2X bit in the UART configuration.
   Contains 0 otherwise.
*/
#define USE_2X 0

#else /* !__DOXYGEN__ */

#undef USE_2X

/* Baud rate tolerance is 2 % unless previously defined */
#ifndef BAUD_TOL
#  define BAUD_TOL 2
#endif

#ifdef __ASSEMBLER__
#define UBRR_VALUE (((F_CPU) + 8 * (BAUD)) / (16 * (BAUD)) -1)
#else
#define UBRR_VALUE (((F_CPU) + 8UL * (BAUD)) / (16UL * (BAUD)) -1UL)
#endif

#if 100 * (F_CPU) > \
  (16 * ((UBRR_VALUE) + 1)) * (100 * (BAUD) + (BAUD) * (BAUD_TOL))
#  define USE_2X 1
#elif 100 * (F_CPU) < \
  (16 * ((UBRR_VALUE) + 1)) * (100 * (BAUD) - (BAUD) * (BAUD_TOL))
#  define USE_2X 1
#else
#  define USE_2X 0
#endif

#if USE_2X
/* U2X required, recalculate */
#undef UBRR_VALUE

#ifdef __ASSEMBLER__
#define UBRR_VALUE (((F_CPU) + 4 * (BAUD)) / (8 * (BAUD)) -1)
#else
#define UBRR_VALUE (((F_CPU) + 4UL * (BAUD)) / (8UL * (BAUD)) -1UL)
#endif

#if 100 * (F_CPU) > \
  (8 * ((UBRR_VALUE) + 1)) * (100 * (BAUD) + (BAUD) * (BAUD_TOL))
#  warning "Baud rate achieved is higher than allowed"
#endif

#if 100 * (F_CPU) < \
```

```
  (8 * ((UBRR_VALUE) + 1)) * (100 * (BAUD) - (BAUD) * (BAUD_TOL))
#  warning "Baud rate achieved is lower than allowed"
#endif

#endif /* USE_U2X */

#ifdef UBRR_VALUE
   /* Check for overflow */
#  if UBRR_VALUE >= (1 « 12)
#    warning "UBRR value overflow"
#  endif

#  define UBRRL_VALUE (UBRR_VALUE & 0xff)
#  define UBRRH_VALUE (UBRR_VALUE » 8)
#endif

#endif /* __DOXYGEN__ */
/* end of util/setbaud.h */
```

23.76 compat/twi.h

```
/* Copyright (c) 2005 Joerg Wunsch
   All rights reserved.

   Redistribution and use in source and binary forms, with or without
   modification, are permitted provided that the following conditions are met:

   * Redistributions of source code must retain the above copyright
     notice, this list of conditions and the following disclaimer.

   * Redistributions in binary form must reproduce the above copyright
     notice, this list of conditions and the following disclaimer in
     the documentation and/or other materials provided with the
     distribution.

   * Neither the name of the copyright holders nor the names of
     contributors may be used to endorse or promote products derived
     from this software without specific prior written permission.

  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
  AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
  IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
  ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
  LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
  CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
  SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
  INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
  CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
  ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
  POSSIBILITY OF SUCH DAMAGE. */

/* $Id$ */

#ifndef _COMPAT_TWI_H_
#define _COMPAT_TWI_H_

#include <util/twi.h>

#endif /* _COMPAT_TWI_H_ */
```

23.77 twi.h File Reference

Macros

TWSR values

Mnemonics:
TW_MT_xxx - master transmitter
TW_MR_xxx - master receiver
TW_ST_xxx - slave transmitter
TW_SR_xxx - slave receiver

- #define TW_START 0x08
- #define TW_REP_START 0x10
- #define TW_MT_SLA_ACK 0x18

- #define TW_MT_SLA_NACK 0x20
- #define TW_MT_DATA_ACK 0x28
- #define TW_MT_DATA_NACK 0x30
- #define TW_MT_ARB_LOST 0x38
- #define TW_MR_ARB_LOST 0x38
- #define TW_MR_SLA_ACK 0x40
- #define TW_MR_SLA_NACK 0x48
- #define TW_MR_DATA_ACK 0x50
- #define TW_MR_DATA_NACK 0x58
- #define TW_ST_SLA_ACK 0xA8
- #define TW_ST_ARB_LOST_SLA_ACK 0xB0
- #define TW_ST_DATA_ACK 0xB8
- #define TW_ST_DATA_NACK 0xC0
- #define TW_ST_LAST_DATA 0xC8
- #define TW_SR_SLA_ACK 0x60
- #define TW_SR_ARB_LOST_SLA_ACK 0x68
- #define TW_SR_GCALL_ACK 0x70
- #define TW_SR_ARB_LOST_GCALL_ACK 0x78
- #define TW_SR_DATA_ACK 0x80
- #define TW_SR_DATA_NACK 0x88
- #define TW_SR_GCALL_DATA_ACK 0x90
- #define TW_SR_GCALL_DATA_NACK 0x98
- #define TW_SR_STOP 0xA0
- #define TW_NO_INFO 0xF8
- #define TW_BUS_ERROR 0x00
- #define TW_STATUS_MASK
- #define TW_STATUS (TWSR & TW_STATUS_MASK)

R/~W bit in SLA+R/W address field.

- #define TW_READ 1
- #define TW_WRITE 0

23.78 util/twi.h

Go to the documentation of this file.

```
/* Copyright (c) 2002, Marek Michalkiewicz
   Copyright (c) 2005, 2007 Joerg Wunsch
   All rights reserved.

   Redistribution and use in source and binary forms, with or without
   modification, are permitted provided that the following conditions are met:

   * Redistributions of source code must retain the above copyright
     notice, this list of conditions and the following disclaimer.

   * Redistributions in binary form must reproduce the above copyright
     notice, this list of conditions and the following disclaimer in
     the documentation and/or other materials provided with the
     distribution.

   * Neither the name of the copyright holders nor the names of
     contributors may be used to endorse or promote products derived
     from this software without specific prior written permission.

  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
  AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
  IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
  ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
  LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
  CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
  SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
  INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
  CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
  ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
  POSSIBILITY OF SUCH DAMAGE. */

/* $Id$ */
/* copied from: Id: avr/twi.h,v 1.4 2004/11/01 21:19:54 arcanum Exp */

```

```
#ifndef _UTIL_TWI_H_
#define _UTIL_TWI_H_ 1

#include <avr/io.h>

/** \file */
/** \defgroup util_twi <util/twi.h>: TWI bit mask definitions
    \code #include <util/twi.h> \endcode

    This header file contains bit mask definitions for use with
    the AVR TWI interface.
*/
/** \name TWSR values

  Mnemonics:
  <br>TW_MT_xxx - master transmitter
  <br>TW_MR_xxx - master receiver
  <br>TW_ST_xxx - slave transmitter
  <br>TW_SR_xxx - slave receiver
  */

/**@{*/
/* Master */
/** \ingroup util_twi
    \def TW_START
    start condition transmitted */
#define TW_START        0x08

/** \ingroup util_twi
    \def TW_REP_START
    repeated start condition transmitted */
#define TW_REP_START        0x10

/* Master Transmitter */
/** \ingroup util_twi
    \def TW_MT_SLA_ACK
    SLA+W transmitted, ACK received */
#define TW_MT_SLA_ACK       0x18

/** \ingroup util_twi
    \def TW_MT_SLA_NACK
    SLA+W transmitted, NACK received */
#define TW_MT_SLA_NACK      0x20

/** \ingroup util_twi
    \def TW_MT_DATA_ACK
    data transmitted, ACK received */
#define TW_MT_DATA_ACK      0x28

/** \ingroup util_twi
    \def TW_MT_DATA_NACK
    data transmitted, NACK received */
#define TW_MT_DATA_NACK     0x30

/** \ingroup util_twi
    \def TW_MT_ARB_LOST
    arbitration lost in SLA+W or data */
#define TW_MT_ARB_LOST      0x38

/* Master Receiver */
/** \ingroup util_twi
    \def TW_MR_ARB_LOST
    arbitration lost in SLA+R or NACK */
#define TW_MR_ARB_LOST      0x38

/** \ingroup util_twi
    \def TW_MR_SLA_ACK
    SLA+R transmitted, ACK received */
#define TW_MR_SLA_ACK       0x40

/** \ingroup util_twi
    \def TW_MR_SLA_NACK
    SLA+R transmitted, NACK received */
#define TW_MR_SLA_NACK      0x48

/** \ingroup util_twi
    \def TW_MR_DATA_ACK
    data received, ACK returned */
#define TW_MR_DATA_ACK      0x50

/** \ingroup util_twi
    \def TW_MR_DATA_NACK
    data received, NACK returned */
#define TW_MR_DATA_NACK     0x58

/* Slave Transmitter */
/** \ingroup util_twi
```

```
    \def TW_ST_SLA_ACK
    SLA+R received, ACK returned */
#define TW_ST_SLA_ACK       0xA8

/** \ingroup util_twi
    \def TW_ST_ARB_LOST_SLA_ACK
    arbitration lost in SLA+RW, SLA+R received, ACK returned */
#define TW_ST_ARB_LOST_SLA_ACK  0xB0

/** \ingroup util_twi
    \def TW_ST_DATA_ACK
    data transmitted, ACK received */
#define TW_ST_DATA_ACK      0xB8

/** \ingroup util_twi
    \def TW_ST_DATA_NACK
    data transmitted, NACK received */
#define TW_ST_DATA_NACK     0xC0

/** \ingroup util_twi
    \def TW_ST_LAST_DATA
    last data byte transmitted, ACK received */
#define TW_ST_LAST_DATA     0xC8

/* Slave Receiver */
/** \ingroup util_twi
    \def TW_SR_SLA_ACK
    SLA+W received, ACK returned */
#define TW_SR_SLA_ACK       0x60

/** \ingroup util_twi
    \def TW_SR_ARB_LOST_SLA_ACK
    arbitration lost in SLA+RW, SLA+W received, ACK returned */
#define TW_SR_ARB_LOST_SLA_ACK  0x68

/** \ingroup util_twi
    \def TW_SR_GCALL_ACK
    general call received, ACK returned */
#define TW_SR_GCALL_ACK     0x70

/** \ingroup util_twi
    \def TW_SR_ARB_LOST_GCALL_ACK
    arbitration lost in SLA+RW, general call received, ACK returned */
#define TW_SR_ARB_LOST_GCALL_ACK 0x78

/** \ingroup util_twi
    \def TW_SR_DATA_ACK
    data received, ACK returned */
#define TW_SR_DATA_ACK      0x80

/** \ingroup util_twi
    \def TW_SR_DATA_NACK
    data received, NACK returned */
#define TW_SR_DATA_NACK     0x88

/** \ingroup util_twi
    \def TW_SR_GCALL_DATA_ACK
    general call data received, ACK returned */
#define TW_SR_GCALL_DATA_ACK    0x90

/** \ingroup util_twi
    \def TW_SR_GCALL_DATA_NACK
    general call data received, NACK returned */
#define TW_SR_GCALL_DATA_NACK   0x98

/** \ingroup util_twi
    \def TW_SR_STOP
    stop or repeated start condition received while selected */
#define TW_SR_STOP      0xA0

/* Misc */
/** \ingroup util_twi
    \def TW_NO_INFO
    no state information available */
#define TW_NO_INFO      0xF8

/** \ingroup util_twi
    \def TW_BUS_ERROR
    illegal start or stop condition */
#define TW_BUS_ERROR        0x00


/**
 * \ingroup util_twi
 * \def TW_STATUS_MASK
 * The lower 3 bits of TWSR are reserved on the ATmega163.
 * The 2 LSB carry the prescaler bits on the newer ATmegas.
```

```
 */
#define TW_STATUS_MASK      (_BV(TWS7)|_BV(TWS6)|_BV(TWS5)|_BV(TWS4)|\
                             _BV(TWS3))
/**
 * \ingroup util_twi
 * \def TW_STATUS
 *
 * TWSR, masked by TW_STATUS_MASK
 */
#define TW_STATUS       (TWSR & TW_STATUS_MASK)
/**@}*/

/**
 * \name R/~W bit in SLA+R/W address field.
 */

/**@{*/
/** \ingroup util_twi
    \def TW_READ
    SLA+R address */
#define TW_READ     1

/** \ingroup util_twi
    \def TW_WRITE
    SLA+W address */
#define TW_WRITE    0
/**@}*/

#endif  /* _UTIL_TWI_H_ */
```

23.79 usa_dst.h File Reference

Functions

- int usa_dst (const time_t ∗timer, int32_t ∗z)

23.80 usa_dst.h

Go to the documentation of this file.

```
/*
 * (c)2012 Michael Duane Rice All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions are
 * met:
 *
 * Redistributions of source code must retain the above copyright notice, this
 * list of conditions and the following disclaimer. Redistributions in binary
 * form must reproduce the above copyright notice, this list of conditions
 * and the following disclaimer in the documentation and/or other materials
 * provided with the distribution. Neither the name of the copyright holders
 * nor the names of contributors may be used to endorse or promote products
 * derived from this software without specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
 * AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
 * LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
 * CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
 * SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
 * INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
 * CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
 * ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
 * POSSIBILITY OF SUCH DAMAGE.
 */

/* $Id$ */

#ifndef USA_DST_H
#define USA_DST_H

#ifdef __cplusplus
extern "C" {
#endif

```

```
#include <time.h>
#include <stdint.h>

/** \file */
/** \defgroup usa_dst <util/usa_dst.h>: Daylight Saving function for the USA.
    \code #include <util/usa_dst.h> \endcode
    Daylight Saving function for the USA. */

/** \ingroup usa_dst
    \fn int usa_dst (const time_t *timer, int32_t *z)
    To utilize this function, call
    \code set_dst(usa_dst); \endcode

    Given the time stamp and time zone parameters provided, the Daylight
    Saving function must return a value appropriate for the tm structures'
    tm_isdst element. That is:

    - \c 0 : If Daylight Saving is not in effect.

    - \c -1 : If it cannot be determined if Daylight Saving is in effect.

    - A positive integer : Represents the number of seconds a clock is
    advanced for Daylight Saving.  This will typically be ONE_HOUR.

    Daylight Saving 'rules' are subject to frequent change. For production
    applications it is recommended to write your own DST function, which
    uses 'rules' obtained from, and modifiable by, the end user
    (perhaps stored in EEPROM).
*/
int usa_dst (const time_t *timer, int32_t *z);

#ifdef __cplusplus
}
#endif

#endif
```

23.81 eedef.h

```
/* Copyright (c) 2009  Dmitry Xmelkov
   All rights reserved.

   Redistribution and use in source and binary forms, with or without
   modification, are permitted provided that the following conditions are met:

   * Redistributions of source code must retain the above copyright
     notice, this list of conditions and the following disclaimer.
   * Redistributions in binary form must reproduce the above copyright
     notice, this list of conditions and the following disclaimer in
     the documentation and/or other materials provided with the
     distribution.
   * Neither the name of the copyright holders nor the names of
     contributors may be used to endorse or promote products derived
     from this software without specific prior written permission.

   THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
   AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
   IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
   ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
   LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
   CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
   SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
   INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
   CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
   ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
   POSSIBILITY OF SUCH DAMAGE.  */

/* $Id$ */

#ifndef _EEDEF_H_
#define _EEDEF_H_  1

#ifndef __DOXYGEN__

/* EEPROM address arg for a set of byte/word/dword functions and for
   the internal eeprom_read_blraw().    */
#define addr_lo r24
#define addr_hi r25

/* Number of bytes arg for all block read/write functions, include
   internal.    */
#define n_lo    r20
#define n_hi    r21

```

```
#if __AVR_XMEGA__

# define NVM_BASE   NVM_ADDR0

#if defined(NVMCTRL_CTRLA)
#  undef NVM_BASE
#  define NVM_BASE       NVMCTRL_CTRLA

#  define NVM_ADDR0      NVMCTRL_ADDR0
#  define NVM_ADDR1      NVMCTRL_ADDR1
#  define NVM_DATA0      NVMCTRL_DATA0
#  define NVM_DATA1      NVMCTRL_DATA1
#  define NVM_NVMBUSY_bp NVMCTRL_EEBUSY_bp
#  define NVM_STATUS     NVMCTRL_STATUS
#  define NVM_CTRLA      NVMCTRL_CTRLA
#  define NVM_CTRLB      NVMCTRL_CTRLB
#  ifndef CCP_SPM_gc
#    define CCP_SPM_gc   (0x9D)
#  endif
#  ifndef NVMCTRL_CMD_PAGEERASEWRITE_gc
#    if NVMCTRL_CMD_gm == 0x7F
#      if defined (__AVR_AVR16EA28__) || defined (__AVR_AVR16EA32__) || defined (__AVR_AVR16EA48__) || \
          defined (__AVR_AVR16EB14__) || defined (__AVR_AVR16EB20__) || defined (__AVR_AVR16EB28__) || \
          defined (__AVR_AVR16EB32__) || defined (__AVR_AVR32EA28__) || defined (__AVR_AVR32EA32__) || \
          defined (__AVR_AVR32EA48__) || defined (__AVR_AVR64EA28__) || defined (__AVR_AVR64EA32__) || \
          defined (__AVR_AVR64EA48__)
         /* AVR-Ex family
          * value of NVMCTRL_CMD_enum.NVMCTRL_CMD_EEPERW_gc */
#        define NVMCTRL_CMD_PAGEERASEWRITE_gc (0x15«0)
#      elif defined (__AVR_AVR32DA28__) || defined (__AVR_AVR32DA32__) || defined (__AVR_AVR32DA48__) || \
            defined (__AVR_AVR64DA28__) || defined (__AVR_AVR64DA32__) || defined (__AVR_AVR64DA48__) || \
            defined (__AVR_AVR64DA64__) || defined (__AVR_AVR128DA28__) || defined (__AVR_AVR128DA32__) || \
            defined (__AVR_AVR128DA48__) || defined (__AVR_AVR128DA64__) || defined (__AVR_AVR32DB28__) || \
            defined (__AVR_AVR32DB32__) || defined (__AVR_AVR32DB48__) || defined (__AVR_AVR64DB28__) || \
            defined (__AVR_AVR64DB32__) || defined (__AVR_AVR64DB48__) || defined (__AVR_AVR64DB64__) || \
            defined (__AVR_AVR128DB28__) || defined (__AVR_AVR128DB32__) || defined (__AVR_AVR128DB48__) || \
            defined (__AVR_AVR128DB64__) || defined (__AVR_AVR16DD14__) || defined (__AVR_AVR16DD20__) || \
            defined (__AVR_AVR16DD28__) || defined (__AVR_AVR16DD32__) || defined (__AVR_AVR32DD14__) || \
            defined (__AVR_AVR32DD20__) || defined (__AVR_AVR32DD32__) || defined (__AVR_AVR32DD28__) || \
            defined (__AVR_AVR64DD14__) || defined (__AVR_AVR64DD20__) || defined (__AVR_AVR64DD28__) || \
            defined (__AVR_AVR64DD32__) \
            || defined (__AVR_AVR64DU28__) || defined (__AVR_AVR64DU32__)
         /* AVR-Dx family
          * value of NVMCTRL_CMD_enum.NVMCTRL_CMD_EEERWR_gc */
#        define NVMCTRL_CMD_PAGEERASEWRITE_gc (0x13«0)
#      else
         /* To add support for a new device, define NVMCTRL CMD_PAGEERASEWRITE_gc with the value
          * of "Erase and Write EEPROM Page" comand code for - Persistent Memory Controller (NVMCTRL).*/
#        error "Not supported devices"
#      endif
#    else
       /* the rest of the AVR devices with NVMCTRL_CTRLA (0x07)
        * value of NVMCTRL_CMD_enum.NVMCTRL_CMD_PAGEERASEWRITE_gc */
#      define NVMCTRL_CMD_PAGEERASEWRITE_gc 3
#    endif
#  endif /* NVMCTRL_CMD_PAGEERASEWRITE_gc */
#endif /* defined(NVMCTRL_CTRLA) */
#else

# if    !defined (EECR) && defined (DEECR)  /* AT86RF401    */
#  define EECR  DEECR
#  define EEARL DEEAR
#  define EEDR  DEEDR
# endif

# if    !defined (EERE) && defined (EER)    /* AT86RF401    */
#  define EERE  EER
# endif

# if    !defined (EEWE) && defined (EEPE)   /* A part of Mega and Tiny */
#  define EEWE  EEPE
```

```
# endif
# if    !defined (EEWE) && defined (EEL)     /* AT86RF401    */
#  define EEWE  EEL
# endif

# if    !defined (EEMWE) && defined (EEMPE) /* A part of Mega and Tiny */
#  define EEMWE EEMPE
# endif
# if    !defined (EEMWE) && defined (EEU)   /* AT86RF401    */
#  define EEMWE EEU
# endif

# if    !_SFR_IO_REG_P (EECR)    \
     || !_SFR_IO_REG_P (EEDR)    \
     || !_SFR_IO_REG_P (EEARL)   \
     || (defined (EEARH) && !_SFR_IO_REG_P (EEARH))
#  error
# endif

#endif  /* !__AVR_XMEGA__ */
#endif  /* !__DOXYGEN__ */
#endif  /* !_EEDEF_H_ */
```

23.82 fdevopen.c File Reference

Functions

- FILE ∗ fdevopen (int(∗put)(char, FILE ∗), int(∗get)(FILE ∗))

23.83 stdio_private.h

```
/* Copyright (c) 2002,2005, Joerg Wunsch
   All rights reserved.

   Redistribution and use in source and binary forms, with or without
   modification, are permitted provided that the following conditions are met:

   * Redistributions of source code must retain the above copyright
     notice, this list of conditions and the following disclaimer.
   * Redistributions in binary form must reproduce the above copyright
     notice, this list of conditions and the following disclaimer in
     the documentation and/or other materials provided with the
     distribution.
   * Neither the name of the copyright holders nor the names of
     contributors may be used to endorse or promote products derived
     from this software without specific prior written permission.

  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
  AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
  IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
  ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
  LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
  CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
  SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
  INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
  CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
  ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
  POSSIBILITY OF SUCH DAMAGE.
*/

/* $Id$ */

#include <stdint.h>
#include <stdio.h>

/* values for PRINTF_LEVEL */
#define PRINTF_MIN 1
#define PRINTF_STD 2
#define PRINTF_FLT 3

/* values for SCANF_LEVEL */
#define SCANF_MIN 1
#define SCANF_STD 2
#define SCANF_FLT 3
```

23.84 xtoa_fast.h

```
/* Copyright (c) 2005, Dmitry Xmelkov
   All rights reserved.

   Redistribution and use in source and binary forms, with or without
   modification, are permitted provided that the following conditions are met:

   * Redistributions of source code must retain the above copyright
     notice, this list of conditions and the following disclaimer.
   * Redistributions in binary form must reproduce the above copyright
     notice, this list of conditions and the following disclaimer in
     the documentation and/or other materials provided with the
     distribution.
   * Neither the name of the copyright holders nor the names of
     contributors may be used to endorse or promote products derived
     from this software without specific prior written permission.

  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
  AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
  IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
  ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
  LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
  CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
  SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
  INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
  CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
  ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
  POSSIBILITY OF SUCH DAMAGE. */

/* $Id$ */

#ifndef _XTOA_FAST_H_
#define _XTOA_FAST_H_

#ifndef __ASSEMBLER__

#include <stddef.h> /* for 'size_t' */

char * itoa_fast (int val, char *s, int base);
char * utoa_fast (unsigned val, char *s, int base);
char * ltoa_fast (long val, char *s, int base);
char * ultoa_fast (unsigned long val, char *s, int base);

char * itoa_width (int val, char *s, int base, size_t width);
char * utoa_width (unsigned val, char *s, int base, size_t width);
char * ltoa_width (long val, char *s, int base, size_t width);
char * ultoa_width (unsigned long val, char *s, int base, size_t width);

/* Internal function for use from `printf'. */
char * __ultoa_invert (unsigned long val, char *s, int base);

#endif  /* ifndef __ASSEMBLER__ */

/* Next flags are to use with `base'. Unused fields are reserved.   */
#define XTOA_PREFIX 0x0100  /* put prefix for octal or hex  */
#define XTOA_UPPER  0x0200  /* use upper case letters   */

#endif  /* _XTOA_FAST_H_ */
```

23.85 dtoa_conv.h

```
/* Copyright (c) 2005, Dmitry Xmelkov
   All rights reserved.

   Redistribution and use in source and binary forms, with or without
   modification, are permitted provided that the following conditions are met:

   * Redistributions of source code must retain the above copyright
     notice, this list of conditions and the following disclaimer.
   * Redistributions in binary form must reproduce the above copyright
     notice, this list of conditions and the following disclaimer in
     the documentation and/or other materials provided with the
     distribution.
   * Neither the name of the copyright holders nor the names of
     contributors may be used to endorse or promote products derived
     from this software without specific prior written permission.

  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
  AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
  IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
  ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
  LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
  CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
```

```
  SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
  INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
  CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
  ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
  POSSIBILITY OF SUCH DAMAGE. */

/* $Id$ */

#ifndef _DTOA_CONV_H
#define _DTOA_CONV_H

#include <stdio.h>

int ftoa_prf (float val, char *s, unsigned char width, unsigned char prec,
              unsigned char flags);

#define DTOA_SPACE  0x01    /* put space for positives  */
#define DTOA_PLUS   0x02    /* put '+' for positives    */
#define DTOA_UPPER  0x04    /* use uppercase letters    */
#define DTOA_ZFILL  0x08    /* fill zeroes          */
#define DTOA_LEFT   0x10    /* adjust to left       */
#define DTOA_NOFILL 0x20    /* do not fill to width     */
#define DTOA_EXP    0x40    /* d2stream: 'e(E)' format  */
#define DTOA_FIX    0x80    /* d2stream: 'f(F)' format  */

#define DTOA_EWIDTH     (-1)    /* Width too small  */
#define DTOA_NONFINITE  (-2)    /* Value is NaN or Inf  */

#endif  /* !_DTOA_CONV_H */
```

23.86 stdlib_private.h

```
/* Copyright (c) 2004, Joerg Wunsch
   All rights reserved.

   Redistribution and use in source and binary forms, with or without
   modification, are permitted provided that the following conditions are met:

   * Redistributions of source code must retain the above copyright
     notice, this list of conditions and the following disclaimer.
   * Redistributions in binary form must reproduce the above copyright
     notice, this list of conditions and the following disclaimer in
     the documentation and/or other materials provided with the
     distribution.
   * Neither the name of the copyright holders nor the names of
     contributors may be used to endorse or promote products derived
     from this software without specific prior written permission.

  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
  AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
  IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
  ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
  LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
  CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
  SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
  INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
  CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
  ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
  POSSIBILITY OF SUCH DAMAGE.
*/

/* $Id$ */

#include <inttypes.h>
#include <stdlib.h>
#include <avr/io.h>

#if !defined(__DOXYGEN__)

struct __freelist {
    size_t sz;
    struct __freelist *nx;
};

#endif

extern char *__brkval;      /* first location not yet allocated */
extern struct __freelist *__flp; /* freelist pointer (head of freelist) */
extern size_t __malloc_margin;   /* user-changeable before the first malloc() */
extern char *__malloc_heap_start;
extern char *__malloc_heap_end;

#ifndef __AVR__

```

```
/*
 * When compiling malloc.c/realloc.c natively on a host machine, it will
 * include a main() that performs a regression test.  This is meant as
 * a debugging aid, where a normal source-level debugger will help to
 * verify that the various allocator structures have the desired
 * appearance at each stage.
 *
 * When cross-compiling with avr-gcc, it will compile into just the
 * library functions malloc() and free().
 */
#define MALLOC_TEST

#endif /* !__AVR__ */

#ifdef MALLOC_TEST

extern void *mymalloc(size_t);
extern void myfree(void *);
extern void *myrealloc(void *, size_t);

#define malloc mymalloc
#define free myfree
#define realloc myrealloc

#define __heap_start mymem[0]
#define __heap_end mymem[256]
extern char mymem[];
#define STACK_POINTER() (mymem + 256)

#else /* !MALLOC_TEST */

extern char __heap_start;
extern char __heap_end;

/* Needed for definition of AVR_STACK_POINTER_REG. */
#include <avr/io.h>

#define STACK_POINTER() ((char *)AVR_STACK_POINTER_REG)

#endif /* MALLOC_TEST */
```

23.87 ephemera_common.h

```
/*
 * (C)2012 Michael Duane Rice All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions are
 * met:
 *
 * Redistributions of source code must retain the above copyright notice, this
 * list of conditions and the following disclaimer. Redistributions in binary
 * form must reproduce the above copyright notice, this list of conditions
 * and the following disclaimer in the documentation and/or other materials
 * provided with the distribution. Neither the name of the copyright holders
 * nor the names of contributors may be used to endorse or promote products
 * derived from this software without specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
 * AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
 * LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
 * CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
 * SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
 * INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
 * CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
 * ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
 * POSSIBILITY OF SUCH DAMAGE.
 */

/* $Id$ */

#ifndef EPHEMERA_PRIVATE_H
#define EPHEMERA_PRIVATE_H

#define TROP_YEAR 31556925
#define ANOM_YEAR 31558433
#define INCLINATION 0.409105176667471    /* Earths axial tilt at the epoch */
#define PERIHELION 31316400    /* perihelion of 1999, 03 jan 13:00 UTC */
#define SOLSTICE 836160        /* winter solstice of 1999, 22 Dec 07:44 UTC */
#define TWO_PI 6.283185307179586
#define TROP_CYCLE 5022440.6025
#define ANOM_CYCLE 5022680.6082
```

```
#define DELTA_V 0.03342044    /* 2x orbital eccentricity */

#endif
```

Index