

GCC-AVR Inline Assembler Cookbook

Version 1.6

About this Document

Maintenance of this document has stopped. Most parts have been integrated into the AVR Libc User's Manual, which can be downloaded from

<http://www.freesoftware.fsf.org/avr-libc/>

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

Because of a lack of documentation, especially for the AVR version of the compiler, it may take some time to figure out the implementation details by studying the compiler and assembler source code. There are also a few sample programs available in the net. Hopefully this document will help to increase their number.

It's assumed, that you are familiar with writing AVR assembler programs, because this is no AVR assembler programming tutorial. It's no C language tutorial either.

Copyright (C) 2001-2002 by egnite Software GmbH

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

This document describes version 3.2 of the compiler. There may be some parts, which hadn't been completely understood by the author himself and not all samples had been tested so far. Because the author is German and not familiar with the English language, there are definitely some typos and syntax errors in the text. As a programmer the author knows, that a wrong documentation sometimes might be worse than none. Anyway, he decided to offer his little knowledge to the public, in the hope to get enough response to improve this document.

Herne, 22nd of August 2002
Harald Kipp
harald.kipp@egnite.de

Contents

1	GCC Statement asm	3
2	Assembler Code	4
3	Input and Output Operands	5
4	Clobbers	7
5	Assembler Macros	9
6	C Stub Functions	9
7	C Names Used in Assembler Code	10
8	Frequently Answered Questions	11
8.1	Using Constants	11
9	Links	12
10	Index	13

History

05/07/01 V 1.1: Output and input lists sequence corrected in chapter 1. Multibyte operands added. Index added. Some typos corrected.

05/07/01 V 1.2: subi replaced by inc in clobber samples. Pointer type is mandatory.

05/07/01 V 1.3: Pointer register modifier added.

05/16/02 V 1.4: Fixed the funny "risk" typo. New chapters C Names, FAQ and Links added.

08/10/02 V 1.5: Fixed vairable name in clobber section. New chapter C stub functions added.

08/22/02 V 1.6: Added a note about integration into the official AVR Libc.

1 GCC Statement asm

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

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

Each asm statement is divided by colons into four parts:

1. The assembler instructions, defined as a single string constant:
"in %0, %1"
2. A list of output operands, separated by commas. Our example uses just one:
"=r" (value)
3. A comma separated list of input operands. Again our example uses one operand only:
"I" (PORTD)
4. Clobbered registers, left empty in our example.

You can write assembler instructions in much the same way like normal assembler programs. However, registers and constants are used in a different way, if they refer to expressions of your C program. The connection between registers and C operands is specified in the second and third part of the asm instruction, resp. the list of input and output operands. The general form is

```
asm(code : output operand list : input operand list : clobber list);
```

In the code section operands are referred by a percent sign followed by a single digit. %0 refers to the first %1 to the second operand and so forth. In example above

%0 refers to "=r" (value) and
%1 refers to "P" (port).

This may still look a little odd now, but the syntax of an operand list will be explained soon. Let us first look to that part of a compiler listing, which may have been generated from our example:

```
        lds r24,value
/* #APP */
        in r24, 12
/* #NOAPP */
        sts value,r24
```

The comments have been added by the compiler to inform the assembler, that the included code has not been generated by the compilation of C statements, but by inline assembler statements. The compiler selected r24, to get the value. It may have selected any other register, though. It may not explicitly load or store the value and it may even decide not to include your assembler code at all. All these decisions are part of the compiler's optimization strategie. For example, if you never use the variable value in the remaining part of the C program, the compiler will most likely remove your code, unless you switched off optimization. To avoid this, you can add the volatile attribute to the asm statement:

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

The last part of the asm instruction, the clobber list, is mainly used to tell the compiler about modifications done by the assembler code. This part may be omitted, all other parts are required, but may be left empty. If your assembler routine won't use any input or output operand, still two colons must follow the assembler code string. A good example is a simple statement to disable interrupts:

```
asm volatile("cli");
```

2 Assembler Code

You can use the same assembler instruction mnemonics, as you'd use with any other AVR assembler. And you can write as many assembler statements into one code string as you like and your flash memory is able to hold.

To make it more readable, you should put each statement on a separate line:

```
asm volatile("nop\n\t"  
            "nop\n\t"  
            "nop\n\t"  
            "nop\n\t"  
            ::);
```

The linefeed and tab characters will make the assembler listing generated by the compiler more readable. It may look a bit odd for the first time, but that's the way the compiler creates its own assembler code.

You may also make use of some special registers.

Symbol	Register
<code>__SREG__</code>	Status register at address 0x3F
<code>__SP_H__</code>	Stack pointer high byte at address 0x3E
<code>__SP_L__</code>	Stack pointer low byte at address 0x3D
<code>__tmp_reg__</code>	Register r0, used for temporary storage
<code>__zero_reg__</code>	Register r1, always zero
<code>_PC_</code>	Program counter

Register r0 may be freely used by your assembler code and need not to be restored at the end of your code. It's a good idea to use `__tmp_reg__` and `__zero_reg__` instead of r0 or r1, just in case a new compiler version might change the register definitions.

Constraint characters may be prepended by a single constraint modifier. Constraints without a modifier specify read-only operands. Modifiers are:

Modifier	Specifies
=	Write-only operand, usually used for all output operands.
+	Read-write operand (not supported by inline assembler)
&	Register should be used for output only

Output operands must be write-only and the C expression result must be an lvalue, which means that they must be valid on the left side of assignments. Note, that the compiler will not check, if they are of reasonable type for the kind of operation used in the assembler instructions.

Input operands are, you guessed it, read-only. But what, if you need the same operand for input and output, as read-write operands are not supported? As stated above, read-write operands are not supported in inline assembler code. But there's another solution. For input operators it is possible to use a single digit in the constraint string. Using digit *n* tells the compiler to use the same register as for the *n*-th operand, starting with zero. Here is an example:

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

This statement will swap the nibbles of an 8-bit variable named `value`. Constraint "0" tells the compiler, to use the same input register as for the first operand. Note however, that this doesn't automatically imply the reverse case. The compiler may choose the same registers for input and output, even if not told to do so. This is no problem in most cases, but may be fatal, if the output operator is modified by the assembler code before the input operator is used. In situation, where your code depends on different registers used for input and output operands, you must add the & constraint modifier to your output operand. The following example demonstrates this problem:

```
asm volatile("in %0,%1"    "\n\t"
            "out %1, %2"   "\n\t"
            : "=&r" (input)
            : "I" (port), "r" (output)
            );
```

In this example an input value is read from a port and then an output value is written to the same port. If the compiler would have chosen the same register for input and output, then the output value would have been destroyed on the first assembler instruction. Fortunately this example uses the & constraint modifier to instruct the compiler not to select any register for the output value, which is used for any of the input operands.

Back to swapping. Here is the code to swap high and low byte of a 16-bit value:

```
asm volatile("mov __tmp_reg__, %A0" "\n\t"
            "mov %A0, %B0"         "\n\t"
            "mov %B0, __tmp_reg__" "\n\t"
            : "=r" (value)
            : "0" (value)
            );
```

First you will notice the usage of register `__tmp_reg__`, which we listed among other special registers in chapter 2. You can use this register without saving its contents. Completely new are those letters A and B in `%A0` and `%B0`. In fact they refer to two different 8-bit registers, both containing a part of value.

Another example to swap bytes of a 32-bit value:

```
asm volatile("mov __tmp_reg__, %A0" "\n\t"  
            "mov %A0, %D0" "\n\t"  
            "mov %D0, __tmp_reg__" "\n\t"  
            "mov __tmp_reg__, %B0" "\n\t"  
            "mov %B0, %C0" "\n\t"  
            "mov %C0, __tmp_reg__" "\n\t"  
            : "=r" (value)  
            : "0" (value)  
            );
```

If operands do not fit into a single register, the compiler will automatically assign enough registers to hold the entire operand. In the assembler code you use %A0 to refer to the lowest byte of the first operand, %A1 to the lowest byte of the second operand and so on. The next byte of the first operand will be %B0, the next byte %C0 and so on.

This also implies, that it is often necessary to cast the type of an input operand to the desired size.

A final problem may arise while using pointer register pairs. If you define an input operand

"e" (ptr)

and the compiler selects register Z (r30:r31), then

%A0 refers to r30 and

%B0 refers to r31.

But both versions will fail during the assembly stage of the compiler, if you explicitly need Z, like in

```
ld r24,Z
```

If you write

```
ld r24, %a0
```

with a lower case a following the percent sign, then the compiler will create the proper assembler line.

4 Clobbers

As stated previously, the last part of the asm statement, the list of clobbers, may be omitted, including the colon separator. However, if you are using registers, which had not been passed as operands, you need to inform the compiler about this. The following example will do an atomic increment. It increments an 8-bit value pointed to by a pointer variable in one go, without being interrupted by an interrupt routine or another thread in a multithreaded environment. Note, that we must use a pointer, because the incremented value needs to be stored before interrupts are enabled.

```
asm volatile(  
    "cli" "\n\t"  
    "ld r24, %a0" "\n\t"  
    "inc r24" "\n\t"  
    "st %a0, r24" "\n\t"  
    "sei" "\n\t"  
    :  
    : "e" (ptr)  
    : "r24"  
);
```

The compiler might produce the following code:

```
cli
ld r24, Z
inc r24
st Z, r24
sei
```

One easy solution to avoid clobbering register r24 is, to make use of the special temporary register `__tmp_reg__` defined by the compiler.

```
asm volatile(
    "cli"                "\n\t"
    "ld __tmp_reg__, %a0" "\n\t"
    "inc __tmp_reg__"    "\n\t"
    "st %a0, __tmp_reg__" "\n\t"
    "sei"                "\n\t"
    :
    : "e" (ptr)
);
```

The compiler is prepared to reload this register next time it uses it. Another problem with the above code is, that it should not be called in code sections, where interrupts are disabled and should be kept disabled, because it will enable interrupts at the end. We may store the current status, but then we need another register. Again we can solve this without clobbering a fixed, but let the compiler select it. This could be done with the help of a local C variable.

```
{
    uint8_t s;
    asm volatile(
        "in %0, __SREG__"    "\n\t"
        "cli"                "\n\t"
        "ld __tmp_reg__, %a1" "\n\t"
        "inc __tmp_reg__"    "\n\t"
        "st %a1, __tmp_reg__" "\n\t"
        "out __SREG__, %0"   "\n\t"
        : "=&r" (s)
        : "e" (ptr)
    );
}
```

Now anything seems correct, but it isn't really. The assembler code modifies the variable, that ptr points to. The compiler will not recognize this and may keep its value in any of the other registers. Not only, that the compiler works with the wrong value, but also the assembler code does, because the C program may have modified the value too, but the compiler didn't update the memory location for optimization reasons. The worst thing you can do in this case:

```
{
    uint8_t s;
    asm volatile(
        "in %0, __SREG__"    "\n\t"
        "cli"                "\n\t"
        "ld __tmp_reg__, %a1" "\n\t"
        "inc __tmp_reg__"    "\n\t"
        "st %a1, __tmp_reg__" "\n\t"
        "out __SREG__, %0"   "\n\t"
        : "=&r" (s)
        : "e" (ptr)
        : "memory"
    );
}
```

The special clobber "memory" informs the compiler, that the assembler code may modify any memory location. It forces the compiler to update all variables, which contents is currently hold in a register before executing the assembler code. And of course, everything has to be reloaded again after this code.

In most situations a much better solution would be to declare the pointer destination itself volatile:

```
volatile uint8_t *ptr;
```

This way, the compiler expects the value pointed to by ptr to be changed and will load it whenever used and store it whenever modified.

Situation, in which you need clobbers, are very rare. In most cases there will be better ways. Clobbered registers will force the compiler to store their values before and reload them after your assembler code. Avoiding them, enables the full optimization power.

5 Assembler Macros

In order to reuse your assembler language parts, it is useful to define them as macros and put them into include files. GCC-AVR comes with a bunch of them, which could be found in the directory avr/include. Using such include files may produce compiler warnings, if they are used in modules, which are compiled in strict ANSI mode. To avoid that, you can write `__asm__` instead of `asm` and `__volatile__` instead of `volatile`. These are equivalent aliases.

Another problem with reused macros arises, if you are using labels. In such cases you may make use of the special pattern `%=`, which is replaced by a unique number on each `asm` statement. The following code had been taken from `avr/include/iomacros.h`:

```
#define loop_until_bit_is_clear(port, bit) \
    __asm__ __volatile__ ( \
        "L_%=: " "sbic %0, %1" "\n\t" \
        "rjmp L_%= " \
        : /* no outputs */ \
        : "I" ((uint8_t)(port)), \
        "I" ((uint8_t)(bit)) \
    )
```

When used for the first time, `L_%=` may be translated to `L_1404`, the next usage might create `L_1405` or whatever. In any case, the labels became unique too.

6 C Stub Functions

Macro definitions will include the same assembler code whenever they are referenced. This may not be acceptable for larger routines. In this case you may define a C stub function, containing nothing else than your assembler code.

```

void delay(uint8_t ms)
{
    uint16_t cnt;

    asm volatile(
        "\n"
        "L_d11%=: "      "\n\t"
        "mov %A0, %A2"   "\n\t"
        "mov %B0, %B2"   "\n"
        "L_d12%=: "      "\n\t"
        "sbiw %A0, 1"    "\n\t"
        "brne L_d12%="   "\n\t"
        "dec %1"         "\n\t"
        "brne L_d11%="   "\n\t"
        : "=&w" (cnt)
        : "r" (ms), "r" (delay_count)
    );
}

```

The purpose of this function is to delay the program execution by a specified number of milliseconds, using a counting loop. The global 16 bit variable `delay_count` must contain the CPU clock frequency in Hertz divided by 4000 and must have been set before calling this routine for the first time.

Like described in the clobber section, the routine uses a local variable to hold a temporary value. Another use for a local variable is a return value. The following function returns a 16 bit value read from two successive port addresses.

```

uint16_t inw(uint8_t port)
{
    uint16_t result;

    asm volatile(
        "in %A0,%1"      "\n\t"
        "in %B0,(%1) + 1"
        : "=r" (result)
        : "I" (port)
    );
    return result;
}

```

7 C Names Used in Assembler Code

By default GCC-AVR uses the same symbolic names of functions or variables in C and assembler code. You can specify a different name for the assembler code by using a special form of the `asm` statement:

```
unsigned long value asm("clock") = 3686400;
```

This statement instructs the compiler to use the symbol name `clock` rather than `value`. This makes sense only for external or static variables, because local variables do not have symbolic names in the assembler code. However, local variables may be held in registers.

With GCC-AVR you can specify the use of a specific register:

```
void Count(void)
{
    register unsigned char counter asm("r3");

    ... some code...
    asm volatile("clr r3");
    ... more code...
}
```

The assembler instruction `clr r3` will clear the variable `counter`. GCC-AVR will not completely reserve the specified register. If the optimizer recognizes, that the variable will not be referenced any longer, the register may be re-used. But the compiler is not able to check whether this register usage conflicts with any predefined register. If you reserve too many registers in this way, the compiler may be even run out of registers during code generation.

In order to change the name of a function you need a prototype declaration, because the compiler will not accept the `asm` keyword in the function definition:

```
extern long Calc(void) asm ("CALCULATE");
```

Calling the function `Calc()` will create assembler instructions to call the function `CALCULATE`.

8 Frequently Answered Questions

8.1 Using Constants

Question: When I do the following:

```
asm volatile("sbi 0x15,0x07");
```

everything is OK! But when I do the same but replacing the address of the port by its label, like:

```
asm volatile("sbi PORTB,0x07");
```

I get a compilation error: "Error: constant value required".

```
asm volatile("sbi PORTB,0x07");
```

Answer: `PORTB` is a precompiler definition included in the processor specific file included in `io-avr.h`. As you may know, the precompiler will not touch strings and `PORTB` is passed to the assembler. One way to avoid this problem is:

```
asm volatile("sbi %0, 0x07" : "I" ((unsigned short)(PORTB)));
```

Note, that `iomacros.h` already provides a `sbi()` macro definition, which can be used in C programs.

9 Links

If you think, that an important link is missing, please contact the author.

A GNU Development Environment for the AVR Microcontroller

Cover the details of the GNU Tools that are specific to the AVR family of processors.

By Rich Neswold.

<http://www.enteract.com/~meswold/avr/>

The AVR-GCC Mailing List

maintained by Jason Kyle.

<http://www.avr1.org/mailman/listinfo/avr-gcc-list/>

AVR-Assembler-Tutorial

Learning AVR assembler language with practical examples. By Gerhard Schmidt.

<http://www.avr-asm-tutorial.net/>

10 Index

`%=` 9

`%0` 3

`%A0` 6

`__asm__` 9

`__SP_H__` 4

`__SP_L__` 4

`__SREG__` 4

`__tmp_reg__` 4, 8

`__volatile__` 9

`__zero_reg__` 4

16-bit value 6

32-bit value 7

8-bit value 5

asm 3

assembler code 4

assembler mnemonics 5

atomic increment 7

C 9, 10

clobbers 7

constants 11

constraint characters 5

constraint modifier 6

internal error 5

interrupt 3, 8

labels 9

local variable 8

lvalue 6

macros 9

memory 9

multibyte operand 7

operands 5

optimization 3, 8

pointer register 5, 7

stack pointer 4

status register 4

stub functions 9

type cast 7

unique number 9

volatile 3, 9