GCC-AVR Inline Assembler Cookbook

Version 1.2

About this Document

The GNU C compiler for Atmel AVR risk 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 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 is an early release of this document. It describes version 2.9.5.2 of the compiler, but most parts are still valid for version 2.9.6. 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 comment in the code sometimes might be worse than none. Anyway, he decided to offer his little knowledge to the public, in the hope to get enough response to improve this document. Feel free to contact the author via e-mail. For the latest release check http://www.egnite.de.

Herne, 7th of May 2001
Harald Kipp
harald.kipp@egnite.de

Contents

History
07/05/01 V 1.1: Output and input lists sequence corrected in chapter 1. Multibyte operands added. Index added. Some typos corrected.
07/05/01 V 1.2: subi replaced by inc in clobber samples. Pointer type is mandantory.

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 devided by colons into four parts:

1. The assembler instructions, defined as a single string constant:
   `"in %0, %1"`

2. A list of output operands, separated by commas. Our example uses just one:
   `"=r" (value)`

3. A comma separated list of input operands. Again our example uses one operand only:
   `"I" (PORTD)`

4. Clobbered registers, left empty in our example.

You can write assembler instructions in much the same way 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 refered 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 explicitely load or store the value and it may even decide not to include your assembler code at all. All these decisions are part of the compiler's optimization 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 seperate line:

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

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

You may also make use of some special registers.

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

3 Input and Output Operands

Each input and output operand is described by a constraint string followed by a C expression in parantheses. GCC-AVR 2.9.5.2 knows the following constraint characters:

Constraint	Used for	Range
a	Simple upper registers	r16 to r23
b	Base pointer registers pairs	y,z
d	Upper register	r16 to r31
e	Pointer register pairs	x,y,z
G	Floating point constant	0.0
I	6-bit positive integer constant	0 to 63
J	6-bit negative integer constant	-63 to 0
K	Integer constant	2 (Program counter?)
L	Integer constant	0
l	Lower registers	r0 to r15
M	8-bit integer constant	0 to 255
N	Integer constant	-1
O	Integer constant	8, 16, 24
P	Integer constant	1
r	Any register	r0 to r31
t	Temporary register	r0
w	Special upper register pairs	r24, r26, r28, r30
x	Pointer register pair X	x (r27:r26)
y	Pointer register pair Y	y (r29:r28)
z	Pointer register pair Z	z (r31:r30)

These definitions seem not to fit properly to the AVR instruction set. The author's assumption is, that this part of the compiler has never been really finished in this version, but that assumption may be wrong. The selection of the proper contraint depends on the range of the constants or registers, which must be acceptable to the AVR instruction they are used with. The C compiler doesn't check any line of your assembler code. But it is able to check the constraint against your C expression. However, if you specify the wrong constraints, then the compiler may silently pass wrong code to the assembler. And, of course, the assembler will fail with some cryptic output or internal errors. For example, if you specify the constraint "r" and you are using this register with an "ori" instruction in your assembler code, then the compiler may select any register. This will fail, if the compiler

chooses r2 to r15. (It will never choose r0 or r1, because these are uses for special purposes.) That's why the correct constraint in that case is "d". On the other hand, if you use the constraint "M", the compiler will take care, that you don't pass anything else but an 8-bit value. Later on we will see, how to pass multibyte expression results to the assembler code.

The following table shows all AVR assembler mnemonics, which require operands, and the related contraints. Because of the improper constraint definitions in version 2.9.5.2, they aren't strict enough. There is, for example, no constraint, which restricts integer constants to the range 0 to 7 for bit set and bit clear operations.

Mnemo	Constraints
adc	r,r
add	r,r
adiw	w,I
and	r,r
andi	d,M
asr	r
bclr	I
bld	r,I
brbc	I,label
brbs	I,label
bset	I
bst	r,I
cbi	I,I
cbr	d,I

com	r
cp	r,r
cpc	r,r
cpi	d,M
cpse	r,r
dec	r
elpm	t,z
eor	r,r
in	r,I
inc	r
ld	r,e
ldd	r,b
ldi	d,M
lds	r,label
lpm	t,z

lsl	r
lsr	r
mov	r,r
mul	r,r
neg	r
or	r,r
ori	d,M
out	I,r
pop	r
push	r
rol	r
ror	r
sbc	r,r
sbci	d,M
sbi	I,I

sbic	I,I
sbiw	w,I
sbr	d,M
sbrc	r,I
sbrs	r,I
ser	d
st	e,r
std	b,r
sts	label,r
sub	r,r
subi	d,M
swap	r

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

Modifier	Specifies
=	Write-only operand
+	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 choosen the same register for input and output, then the output value would have been

destroyed on the first assembler instruction. Fortunately this example uses the & constraint modifier to instruct the compiler not to select any register for the output value, which is used for any of the input operands.
Back to swapping. Here is the code to swap high and low byte of a 16-bit value:

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

First you will notice the usage of register __tmp_reg__, which we listed among other special registers in 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)
            );
```

Now the explanation of the additional letters becomes trivial. If the operand assigned to an 8-bit register is a multibyte operand, then 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 might be sometimes neccessary to cast the type of an input operand to the desired size.

4 Clobbers

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

```
asm volatile(
    "cli"               "\n\t"
    "ld r24, %a0"       "\n\t"
    "inc r24"           "\n\t"
    "st %a0, r24"       "\n\t"
    "sei"               "\n\t"
    :
    : "z" (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"
    :
    : "z" (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" (t)
        : "z" (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" (t)
        : "z" (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 Index