ARM GCC Inline Assembler Cookbook

About this Document

The GNU C compiler for ARM 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.

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

This document describes version 3.4 of the compiler.

GCC asm Statement

Let's start with a simple example of rotating bits. It takes the value of one integer variable, right rotates the bits by one and stores the result in a second integer variable.

```
asm("mov %0, %1, ror #1" : "=r" (result) : "r" (value));
```

Each *asm* statement is devided by colons into up to four parts:

1. The assembler instructions, defined as a single string constant:

   ```
   "mov %0, %1, ror #1"
   ```

2. A list of output operands, separated by commas. Our example uses just one:

   ```
   "=r" (result)
   ```

3. A comma separated list of input operands. Again our example uses one operand only:

   ```
   "r" (value)
   ```

4. Clobbered registers, left empty in our example.

You can write assembler instructions in much the same way as you would write 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, the list of output and input operands, respectively. The general form is

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

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

%0 refers to *"=r" (result)* and
%1 refers to *"r" (value)*

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

```
00309DE5                ldr     r3, [sp, #0]    @ value, value
E330A0E1                mov r3, r3, ror #1      @ tmp69, value
04308DE5                str     r3, [sp, #4]    @ tmp71, result
```

The compiler selected register *r3* for bit rotation. It could 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 strategy. 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.

You can add the volatile attribute to the *asm* statement to instruct the compiler not to optimize your assembler code.

```
asm volatile("mov %0, %1, ror #1" : "=r" (result) : "r" (value));
```

As with the clobber list in our example, trailing parts of the asm statement may be omitted, if unused. The following statement does nothing but consuming CPU time and provides the code part only. It is also known as a NOP (no operation) statement and is typically used for tiny delays.

```
asm volatile ("mov r0, r0");
```

If an unused part is followed by one which is used, it must be left empty. The following example uses an input, but no output value.

```
asm volatile ("msr cpsr, %0" : : "r" (status));
```

Even the code part may be left empty, though an empty string is reuired. The next statement specifies a special clobber to tell the compiler, that memory contents may have changed.

```
asm volatile ("" : : : "memory");
```

With inline assembly you can use the same assembler instruction mnemonics as you'd use for writing pure ARM assemly code. And you can write more than one assembler instruction in a single inline asm statement. To make it more readable, you should put each instruction on a separate line.

```
asm volatile(
    "mov     r0, r0\n\t"
    "mov     r0, r0\n\t"
```

```
   "mov      r0, r0\n\t"
   "mov      r0, r0"
);
```

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. Also note, that eight characters are reserved for the assembler instruction mnemonic.

Input and Output Operands

Each input and output operand is described by a constraint string followed by a C expression in parantheses. For ARM processors, GCC 3.4 provides the following constraint characters.

Constraint	Used for	Range
f	Floating point registers	
I	Immediate operands	8 bits, possibly shifted.
J	Indexing constants	-4095 .. 4095
K	Negated value in rhs	-4095 .. 4095
L	Negative value in rhs	-4095 .. 4095
M	For shifts.	0..32 or power of 2
r	General registers	

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, 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 the operands must be valid on the left side of assignments. Note, that the compiler will not check if the operands are of reasonable type for the kind of operation used in the assembler instructions.

Input operands are, you guessed it, read-only. Never ever write to an input operand. But what if you need the same operand for input and output? As stated above, read-write operands are not supported in inline assembler code. But there is 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("mov %0, %0, ror #1" : "=r" (value) : "0" (value));
```

This is similar to our initial example. It rotates the contents of the variable *value* to the right by one bit. In opposite to our first example, the result is not stored in another variable. Instead the original contents of input variable will be modified. Constraint *"0"* tells the compiler, to use the same input register as for the first output 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. In our initial example it did indeed choose the same register r3.

This is not a 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 situations 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("ldr     %0, [%1]"          "\n\t"
             "str     %2, [%1, #4]"      "\n\t"
             : "=&r" (rdv)
             : "r" (&table), "r" (wdv)
             : "memory"
            );
```

In this example a value is read from a table and then another value is written to another location in this table. 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:

Clobbers

If you are using registers, which had not been passed as operands, you need to inform the compiler about this. The following code will adjust a value to a multiple of four. It uses r3 as a scratch register and lets the compiler know about this by specifying r3 in the clobber list. Furthermore the CPU status flags are modified by the *ands* instruction. Adding the pseudo register *cc* to the clobber list will keep the compiler informed about this modification as well.

```
asm volatile("ands    r3, %1, #3"      "\n\t"
             "eor     %0, %0, r3"      "\n\t"
             "addne   %0, #4"
             : "=r" (len)
             : "0" (len)
             : "cc", "r3"
            );
```

Our previous example, which stored a value in a table

```
asm volatile("ldr     %0, [%1]"          "\n\t"
             "str     %2, [%1, #4]"      "\n\t"
             : "=&r" (rdv)
             : "r" (&table), "r" (wdv)
```

```
                    : "memory"
                );
```

uses another so called pseudo register named *"memory"* in the clobber list. This special clobber informs the compiler that the assembler code may modify any memory location. It forces the compiler to update all variables for which the contents are currently held in a register before executing the assembler code. And of course, everything has to be reloaded again after this code.

Assembler Macros

In order to reuse your assembler language parts, it is useful to define them as macros and put them into include files. Nut/OS comes with some of them, which could be found in the subdirectory *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.

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 other than your assembler code.

```
unsigned long htonl(unsigned long val)
{
    asm volatile ("eor r3, %1, %1, ror #16\n\t"
                  "bic r3, r3, #0x00FF0000\n\t"
                  "mov %0, %1, ror #8\n\t"
                  "eor %0, %0, r3, lsr #8"
                  : "=r" (val)
                  : "0"(val)
                  : "r3"
        );
    return val;
}
```

The purpose of this function is to swap all bytes of an unsigend 32 bit value. In other words, it changes a big endian to a little endian value or vice versa.

C Names Used in Assembler Code

By default *GCC* 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* you can further demand the use of a specific register:

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

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

The assembler instruction, *"eor r3, r3, r3"*, will clear the variable counter. Be warned, that this sample is bad in most situations, because it interfers with the compiler's optimizer. Furthermore, *GCC* 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 wether this register usage conflicts with any predefined register. If you reserve too many registers in this way, the compiler may 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*.

Register Usage

Typically the following registers are used by the compiler for specific purposes.

Register	Alt. Name	Usage
r0	a1	First function argument Integer function result Scratch register
r1	a2	Second function argument Scratch register
r2	a3	Third function argument Scratch register
r3	a4	Fourth function argument Scratch register
r4	v1	Register variable
r5	v2	Register variable
r6	v3	Register variable
r7	v4	Register variable
r8	v5	Register variable
r9	v6	Register variable

	rfp	Real frame pointer
r10	sl	Stack limit
r11	fp	Argument pointer
r12	ip	Temporary workspace
r13	sp	Stack pointer
r14	lr	Link register Workspace
r15	pc	Program counter

Links

For a more thorough discussion of inline assembly usage, see the gcc user manual. The latest version of the gcc manual is always available here: http://gcc.gnu.org/onlinedocs/