About this document

(fuente: http://www.ethernut.de/en/documents/arm-inline-asm.html)

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.

All samples had been tested with GCC version 4, but most of them should work with earlier versions too.

GCC asm statement

Let's start with a simple example. The following statement may be included in your code like any other C statement.

```
/* NOP example */
asm("mov r0,r0");
```

It moves the contents of register r0 to register r0. In other words, it doesn't do much more than nothing. It is also known as a NOP (no operation) statement and is typically used for very short delays.

Stop! Before adding this example right away to your C code, keep on reading and learn, why this may not work as expected.

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

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

The special sequence of linefeed and tab characters will keep the assembler listing looking nice. It may seem a bit odd for the first time, but that's the way the compiler creates its own assembler code while compiling C statements.

So far, the assembler instructions are much the same as they'd appear in pure assembly language programs. However, registers and constants are specified in a different way, if they refer to C expressions. The general form of an inline assembler statement is

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

The connection between assembly language and C operands is provided by an optional second and third part of the *asm* statement, the list of output and input operands. We will explain the third optional part, the list of clobbers, later.

The next example of rotating bits passes C variables to assembly language. It takes the value of one integer variable, right rotates the bits by one and stores the result in a second integer variable.

```
/* Rotating bits example */
asm("mov %[result], %[value], ror #1" : [result] "=r" (y) : [value] "r"
(x));
```

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

1. The assembler instructions, defined in a single string literal:

   ```
   "mov %[result], %[value], ror #1"
   ```

2. An optional list of output operands, separated by commas. Each entry consists of a symbolic name enclosed in square brackets, followed by a constraint string, followed by a C expression enclosed in parentheses. Our example uses just one entry:

   ```
   [result] "=r" (y)
   ```

3. A comma separated list of input operands, which uses the same syntax as the list of output operands. Again, this is optional and our example uses one operand only:

   ```
   [value] "r" (x)
   ```

4. Optional list of clobbered registers, omitted in our example.

As shown in the initial NOP example, trailing parts of the asm statement may be omitted, if unused. Inline asm statements containing assembler instruction only are also known as basic inline assembly, while statements containing optional parts are called extended inline assembly. If an unused part is followed by one which is used, it must be left empty. The following example sets the current program status register of the ARM CPU. It uses an input, but no output operand.

```
asm("msr cpsr,%[ps]" : : [ps]"r"(status));
```

Even the code part may be left empty, though an empty string is required. The next statement creates a special clobber to tell the compiler, that memory contents may have changed. Again, the clobber list will be explained later, when we take a look to code optimization.

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

You can insert spaces, newlines and even C comments to increase readability.

```
asm("mov    %[result], %[value], ror #1"


           : [result]"=r" (y) /* Rotation result. */
           : [value]"r"   (x) /* Rotated value. */
           : /* No clobbers */
    );
```

In the code section, operands are referenced by a percent sign followed by the related symbolic name enclosed in square brackets. It refers to the entry in one of the operand lists that contains the same symbolic name. From the rotating bits example:

%[result] refers to output operand, the C variable *y*, and
%[value] refers to the input operand, the C variable *x*.

Symbolic operand names use a separate name space. That means, that there is no relation to any other symbol table. To put it simple: You can choose a name without taking care whether the same name already exists in your C code. However, unique symbols must be used within each asm statement.

If you already looked to some working inline assembler statements written by other authors, you may have noticed a significant difference. In fact, the GCC compiler supports symbolic names since version 3.1. For earlier releases the rotating bit example must be written as

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

Operands are referenced by a percent sign followed by a single digit, where *%0* refers to the first *%1* to the second operand and so forth. This format is still supported by the latest GCC releases, but quite error-prone and difficult to maintain. Imagine, that you have written a large number of assembler instructions, where operands have to be renumbered manually after inserting a new output operand.

If all this stuff still looks a little odd, don't worry. Beside the mysterious clobber list, you have the strong feeling that something else is missing, right? Indeed, we didn't talk about the constraint strings in the operand lists. I'd like to ask for your patience. There's something more important to highlight in the next chapter.

Input and output operands

We learned, that each input and output operand is described by a symbolic name enclosed in square bracket, followed by a constraint string, which in turn is followed by a C expression in parentheses.

What are these constraints and why do we need them? You probably know that every assembly instruction accepts specific operand types only. For example, the *branch* instruction expects a target address to jump at. However, not every memory address is valid, because the final opcode accepts a 24-bit offset only. In contrary, the *branch and exchange* instruction expects a register that contains a 32-bit target address. In both cases the operand passed from C to the inline assembler may be the same C function pointer. Thus, when passing constants, pointers or variables to inline assembly statements, the inline assembler must know, how they should be represented in the assembly code.

For ARM processors, GCC 4 provides the following constraints.

Constraint	**Usage in ARM state**	**Usage in Thumb state**
f	Floating point registers f0 .. f7	Not available
h	Not available	Registers r8..r15
G	Immediate floating point constant	Not available
H	Same a G, but negated	Not available
I	Immediate value in data processing instructions e.g. ORR R0, R0, #operand	Constant in the range 0 .. 255 e.g. SWI operand
J	Indexing constants -4095 .. 4095 e.g. LDR R1, [PC, #operand]	Constant in the range -255 .. -1 e.g. SUB R0, R0, #operand
K	Same as I, but inverted	Same as I, but shifted
L	Same as I, but negated	Constant in the range -7 .. 7 e.g. SUB R0, R1, #operand
l	Same as r	Registers r0..r7 e.g. PUSH operand
M	Constant in the range of 0 .. 32 or a power of 2 e.g. MOV R2, R1, ROR #operand	Constant that is a multiple of 4 in the range of 0 .. 1020 e.g. ADD R0, SP, #operand
m	Any valid memory address	
N	Not available	Constant in the range of 0 .. 31 e.g. LSL R0, R1, #operand
O	Not available	Constant that is a multiple of 4 in the range of -508 .. 508 e.g. ADD SP, #operand
r	General register r0 .. r15 e.g. SUB operand1, operand2, operand3	Not available
w	Vector floating point registers s0 .. s31	Not available
X	Any operand	

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, must be listed as an output operand
&	A register that 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. The C compiler is able to check this.

Input operands are, you guessed it, read-only. Note, that the C compiler will not be able to check, whether the operands are of reasonable type for the kind of operation used in the assembler instructions. Most problems will be detected during the late assembly stage, which is well known for its weird error messages. Even if it claims to have found an internal compiler problem that should be immediately reported to the authors, you better check your inline assembler code first.

A strict rule is: Never ever write to an input operand. But what, if you need the same operand for input and output? The constraint modifier + does the trick as shown in the next example:

```
asm("mov %[value], %[value], ror #1" : [value] "+r" (y));
```

This is similar to our rotating bits example presented above. It rotates the contents of the variable *value* to the right by one bit. In opposite to the previous example, the result is not stored in another variable. Instead the original contents of input variable will be modified.

The modifier + may not be supported by earlier releases of the compiler. Luckily they offer another solution, which still works with the latest compiler version. 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("mov %0, %0, ror #1" : "=r" (value) : "0" (value));
```

Constraint *"0"* tells the compiler, to use the same input register that is used 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. You may remember the first assembly listing of the rotating bits example with two variables, where the compiler used the same register r3 for both variables. The asm statement

```
asm("mov %[result],%[value],ror #1":[result] "=r" (y):[value] "r" (x));
```

generated this code:

```
00309DE5    ldr   r3, [sp, #0]    @ x, x
```

```
E330A0E1    mov   r3, r3, ror #1  @ tmp, x
04308DE5    str   r3, [sp, #4]    @ tmp, y
```

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 code demonstrates this problem.

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

A value is read from a table and then another value is written to another location in this table. 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, the & modifier instructs the compiler not to select any register for the output value, which is used for any of the input operands.