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 assume GCC version 4, but will work with earlier versions as well.

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.

C code optimization

There are two possible reasons why you want to use assembly language. First is, that C is limited when we are getting closer to the hardware. E.g. there's no C statement for directly modifying the processor status register. The second reason is to create highly optimized code. No doubt, the GNU C code optimizer does a good job, but the results are far away from handcrafted assembler code.

The subject of the chapter is often overlooked: When adding assembly language code by using inline assembler statements, this code is also processed by the C compiler's code optimizer. Let's examine the part of a compiler listing which may have been generated from our rotating bits example:

```
00309DE5    ldr   r3, [sp, #0]    @ x, x
E330A0E1    mov   r3, r3, ror #1  @ tmp, x
04308DE5    str   r3, [sp, #4]    @ tmp, y
```

The compiler selected register *r3* for bit rotation. It could have selected any other register or two registers, one for each C variable. It may not explicitly load the value or store the result. Here is another listing, generated by a different compiler version with different compile options:

```
E420A0E1    mov r2, r4, ror #1    @ y, x
```

The compiler selected a unique register for each operand, using the value already cached in r4 and passing the result to the following code in r2. Did you get the picture?

Often it becomes worse. The compiler may even decide not to include your assembler code at all. These decisions are part of the compiler's optimization strategy and depend on the context in which your assembler instructions are used. For example, if you never use any of the output operands in the remaining part of the C program, the optimizer will most likely remove

your inline assembler statement. The NOP example we presented initially may be such a candidate as well, because to the compiler this is useless overhead, slowing down program execution.

The solution is to add the volatile attribute to the *asm* statement to instruct the compiler to exclude your assembler code from code optimization. Remember, that you have been warned to use the initial example. Here is the revised version:

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

But there is more trouble waiting for us. A sophisticated optimizer will re-arrange the code. The following C snippet had been left over after several last minute changes:

```
i++;
if (j == 1)
    x += 3;
i++;
```

The optimizer will recognize, that the two increments do not have any impact on the conditional statement. Furthermore it knows, that incrementing a value by 2 will cost one ARM instruction only. Thus, it will re-arrange the code to

```
if (j == 1)
    x += 3;
i += 2;
```

and save one ARM instruction. As a result: There is no guarantee, that the compiled code will retain the sequence of statements given in the source code.

This may have a great impact on your code, as we will demonstrate now. The following code intends to multiply c with b, of which one or both may be modified by an interrupt routine. Disabling interrupts before accessing the variables and re-enable them afterwards looks like a good idea.

```
asm volatile("mrs r12, cpsr\n\t"
    "orr r12, r12, #0xC0\n\t"
    "msr cpsr_c, r12\n\t" ::: "r12", "cc");
c *= b; /* This may fail. */
asm volatile("mrs r12, cpsr\n"
    "bic r12, r12, #0xC0\n"
    "msr cpsr_c, r12" ::: "r12", "cc");
```

Unfortunately the optimizer may decide to do the multiplication first and then execute both inline assembler instructions or vice versa. This will make our assembly code useless.

We can solve this with the help of the clobber list, which will be explained now. The clobber list from the example above

```
"r12", "cc"
```

informs the compiler that the assembly code modifies register r12 and updates the condition code flags. Btw. using a hard coded register will typically prevent best optimization results. In general you should pass a variable and let the compiler choose the adequate register. Beside register names and *cc* for the condition register, *memory* is a valid keyword too. It tells the compiler that the assembler instruction may change memory locations. This forces the compiler to store all cached values before and reload them after executing the assembler instructions. And it must retain the sequence, because the

contents of all variables is unpredictable after executing an asm statement with a memory clobber.

```
asm volatile("mrs r12, cpsr\n\t"
    "orr r12, r12, #0xC0\n\t"
    "msr cpsr_c, r12\n\t" :: : "r12", "cc", "memory");
c *= b; /* This is safe. */
asm volatile("mrs r12, cpsr\n"
    "bic r12, r12, #0xC0\n"
    "msr cpsr_c, r12" ::: "r12", "cc", "memory");
```

Invalidating all cached values may be suboptimal. Alternatively you can add a dummy operand to create an artificial dependency:

```
asm volatile("mrs r12, cpsr\n\t"
    "orr r12, r12, #0xC0\n\t"
    "msr cpsr_c, r12\n\t" : "=X" (b) :: "r12", "cc");
c *= b; /* This is safe. */
asm volatile("mrs r12, cpsr\n"
    "bic r12, r12, #0xC0\n"
    "msr cpsr_c, r12" :: "X" (c) : "r12", "cc");
```

This code pretends to modify variable b in the first asm statement and to use the contents variable c in the second. This will preserve the sequence of our three statements without invalidating other cached variables.

It is essential to understand how the optimizer affects inline assembler statements. If something remains nebulous, better re-read this part before moving on the the next topic.

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.

More recipes

Inline assembler as preprocessor macro

In order to reuse your assembler language parts, it is useful to define them as macros and put them into include files. 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. Here is a macro which will convert a long value from little endian to big endian or vice versa:

```
#define BYTESWAP(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", "cc" \
    );
```

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. Here is the byte swap procedure again, this time implemented as a C function.

```
```

```
unsigned long ByteSwap(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;
}
```

Replacing symbolic names of C variables

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 symbolic name clock rather than value. This makes sense only for global variables. Local variables (aka auto variables) do not have symbolic names in assembler code.

Replacing symbolic names of C functions

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*.

Forcing usage of specific registers

A local variable may be held in a register. You can instruct the inline assembler to use a specific register for it.

```
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 interferes 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 whether 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.

Using registers temporarily

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 and thus cc had been added to the clobbers.

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

Again, hard coding register usage is always bad coding style. Better implement a C stub function and use a local variable for temporary values.

Register Usage

It is always a good idea to analyze the assembly listing output of the C compiler and study the generated code. The following table of the compiler's typical register usage will be probably helpful to understand the code.

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 rfp	Register variable 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

Common pitfalls

Instruction sequence

Developers often expect, that a sequence of instructions remains in the final code as specified in the source code. This assumption is wrong and often introduces hard to find bugs. Actually, asm statements are processed by the optimizer in the same way as other C statements. They may be rearranged if dependencies allow this.

The chapter "C code optimization" discusses the details and offers solutions.

Executing in Thumb status

Be aware, that, depending on the given compile options, the compiler may switch to thumb state. Using inline assembler with instructions that are not available in thumb state will result in cryptic compile errors.

Assembly code size

In most cases the compiler will correctly determine the size of the assembler instruction, but it may become confused by assembler macros. Better avoid them.

In case you are confused: This is about assembly language macros, not C preprocessor macros. It is fine to use the latter.

Labels

Within the assembler instruction you can use labels as jump targets. However, you must not jump from one assembler instruction into another. The optimizer knows nothing about those branches and may generate bad code.

External 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/

Copyright

As you may (or may not) know, all original work is subject to a copyright, even if not explicitly stated. The previous version of this document had been copied and re-published often, which is great. Some copies claim however, that the publisher (not me) is the original author (grumble). So I decided to publish this document under the following copyright: