# Brennan's Guide to Inline Assembly

by Brennan "Bas" Underwood

Document version 1.1.2.2

Ok. This is meant to be an introduction to inline assembly under DJGPP. DJGPP is based on GCC, so it uses the AT&T/UNIX syntax and has a somewhat unique method of inline assembly. I spent many hours figuring some of this stuff out and told Info that I hate it, many times.

Hopefully if you already know Intel syntax, the examples will be helpful to you. I've put variable names, register names and other literals in bold type.

## The Syntax

So, DJGPP uses the AT&T assembly syntax. What does that mean to you?

- Register naming:
  Register names are prefixed with "%". To reference eax:

  ```
  AT&T:  %eax
  Intel: eax
  ```

- Source/Destination Ordering:
  In AT&T syntax (which is the UNIX standard, BTW) the source is always on the left, and the destination is always on the right.
  So let's load ebx with the value in eax:

  ```
  AT&T:  movl %eax, %ebx
  Intel: mov ebx, eax
  ```

- Constant value/immediate value format:
  You must prefix all constant/immediate values with "$".
  Let's load eax with the address of the "C" variable booga, which is static.

  ```
  AT&T:  movl $_booga, %eax
  Intel: mov eax, _booga
  ```

  Now let's load ebx with 0xd00d:

  ```
  AT&T:  movl $0xd00d, %ebx
  Intel: mov ebx, d00dh
  ```

- Operator size specification:
  You must suffix the instruction with one of b, w, or l to specify the width of the destination register as a byte, word or longword. If you omit this, GAS (GNU assembler) will attempt to guess. You don't want GAS to guess, and guess wrong! Don't forget it.

  ```
  AT&T:  movw %ax, %bx
  Intel: mov bx, ax
  ```

  The equivalent forms for Intel is byte ptr, word ptr, and dword ptr, but that is for when you are...
- Referencing memory:

DJGPP uses 386-protected mode, so you can forget all that real-mode addressing junk, including the restrictions on which register has what default segment, which registers can be base or index pointers. Now, we just get 6 general purpose registers. (7 if you use ebp, but be sure to restore it yourself or compile with -fomit-frame-pointer.)
Here is the canonical format for 32-bit addressing:

```
AT&T:  immed32(basepointer,indexpointer,indexscale)
Intel: [basepointer + indexpointer*indexscale + immed32]
```

You could think of the formula to calculate the address as:

```
  immed32 + basepointer + indexpointer * indexscale
```

You don't have to use all those fields, but you do have to have at least 1 of immed32, basepointer and you MUST add the size suffix to the operator!
Let's see some simple forms of memory addressing:

- Addressing a particular C variable:

  ```
  AT&T:  _booga
  Intel: [_booga]
  ```

  Note: the underscore ("_") is how you get at static (global) C variables from assembler. This only works with global variables. Otherwise, you can use extended asm to have variables preloaded into registers for you. I address that farther down.

- Addressing what a register points to:

  ```
  AT&T:  (%eax)
  Intel: [eax]
  ```

- Addressing a variable offset by a value in a register:

  ```
  AT&T: _variable(%eax)
  Intel: [eax + _variable]
  ```

- Addressing a value in an array of integers (scaling up by 4):

  ```
  AT&T:  _array(,%eax,4)
  Intel: [eax*4 + array]
  ```

- You can also do offsets with the immediate value:

  ```
  C code: *(p+1) where p is a char *
  AT&T:  1(%eax) where eax has the value of p
  Intel: [eax + 1]
  ```

- You can do some simple math on the immediate value:

  ```
  AT&T: _struct_pointer+8
  ```

  I assume you can do that with Intel format as well.

- Addressing a particular char in an array of 8-character records:
  eax holds the number of the record desired. ebx has the wanted char's

offset within the record.

```
AT&T:  _array(%ebx,%eax,8)
Intel: [ebx + eax*8 + _array]
```

Whew. Hopefully that covers all the addressing you'll need to do. As a note, you can put esp into the address, but only as the base register.

## Basic inline assembly

The format for basic inline assembly is very simple, and much like Borland's method.

```
asm ("statements");
```

Pretty simple, no? So

```
asm ("nop");
```

will do nothing of course, and

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

will stop interrupts, with

```
asm ("sti");
```

of course enabling them. You can use __asm__ instead of asm if the keyword asm conflicts with something in your program.

When it comes to simple stuff like this, basic inline assembly is fine. You can even push your registers onto the stack, use them, and put them back.

```
asm ("pushl %eax\n\t"
     "movl $0, %eax\n\t"
     "popl %eax");
```

(The \n's and \t's are there so the .s file that GCC generates and hands to GAS comes out right when you've got multiple statements per asm.)
It's really meant for issuing instructions for which there is no equivalent in C and don't touch the registers.

But if you do touch the registers, and don't fix things at the end of your asm statement, like so:

```
asm ("movl %eax, %ebx");
asm ("xorl %ebx, %edx");
asm ("movl $0, _booga");
```

then your program will probably blow things to hell. This is because GCC hasn't been told that your asm statement clobbered ebx and edx and booga, which it might have been keeping in a register, and might plan on using later. For that, you need:

## Extended inline assembly

The basic format of the inline assembly stays much the same, but now gets Watcom-like extensions to allow input arguments and output arguments.

Here is the basic format:

```
asm ( "statements" : output_registers : input_registers : clobbered_registers);
```

Let's just jump straight to a nifty example, which I'll then explain:

```
asm ("cld\n\t"
     "rep\n\t"
     "stosl"
     : /* no output registers */
     : "c" (count), "a" (fill_value), "D" (dest)
     : "%ecx", "%edi" );
```

The above stores the value in fill_value count times to the pointer dest.

Let's look at this bit by bit.

```
asm ("cld\n\t"
```

We are clearing the direction bit of the flags register. You never know what this is going to be left at, and it costs you all of 1 or 2 cycles.

```
     "rep\n\t"
     "stosl"
```

Notice that GAS requires the rep prefix to occupy a line of it's own. Notice also that stos has the l suffix to make it move longwords.

```
     : /* no output registers */
```

Well, there aren't any in this function.

```
     : "c" (count), "a" (fill_value), "D" (dest)
```

Here we load ecx with count, eax with fill_value, and edi with dest. Why make GCC do it instead of doing it ourselves? Because GCC, in its register allocating, might be able to arrange for, say, fill_value to already be in eax. If this is in a loop, it might be able to preserve eax thru the loop, and save a movl once per loop.

```
     : "%ecx", "%edi" );
```

And here's where we specify to GCC, "you can no longer count on the values you loaded into ecx or edi to be valid." This doesn't mean they will be reloaded for certain. This is the clobberlist.

Seem funky? Well, it really helps when optimizing, when GCC can know exactly what you're doing with the registers before and after. It folds your assembly code into the code it's generates (whose rules for generation look remarkably like the above) and then optimizes. It's even smart enough to know that if you tell it to put (x+1) in a register, then if you don't clobber it, and later C code refers to (x+1), and it was able to keep that register free, it will reuse the computation. Whew.

Here's the list of register loading codes that you'll be likely to use:

```
a       eax
b       ebx
c       ecx
d       edx
S       esi
```

```
D       edi
I       constant value (0 to 31)
q,r     dynamically allocated register (see below)
g       eax, ebx, ecx, edx or variable in memory
A       eax and edx combined into a 64-bit integer (use long longs)
```

Note that you can't directly refer to the byte registers (ah, al, etc.) or the word registers (ax, bx, etc.) when you're loading this way. Once you've got it in there, though, you can specify ax or whatever all you like.

The codes have to be in quotes, and the expressions to load in have to be in parentheses.

When you do the clobber list, you specify the registers as above with the %. If you write to a variable, you must include "memory" as one of The Clobbered. This is in case you wrote to a variable that GCC thought it had in a register. This is the same as clobbering all registers. While I've never run into a problem with it, you might also want to add "cc" as a clobber if you change the condition codes (the bits in the flags register the jnz, je, etc. operators look at.)

Now, that's all fine and good for loading specific registers. But what if you specify, say, ebx, and ecx, and GCC can't arrange for the values to be in those registers without having to stash the previous values. It's possible to let GCC pick the register(s). You do this:

```
asm ("leal (%1,%1,4), %0"
     : "=r" (x)
     : "0" (x) );
```

The above example multiplies x by 5 really quickly (1 cycle on the Pentium). Now, we could have specified, say eax. But unless we really need a specific register (like when using rep movsl or rep stosl, which are hardcoded to use ecx, edi, and esi), why not let GCC pick an available one? So when GCC generates the output code for GAS, %0 will be replaced by the register it picked.

And where did "q" and "r" come from? Well, "q" causes GCC to allocate from eax, ebx, ecx, and edx. "r" lets GCC also consider esi and edi. So make sure, if you use "r" that it would be possible to use esi or edi in that instruction. If not, use "q".

Now, you might wonder, how to determine how the %n tokens get allocated to the arguments. It's a straightforward first-come-first-served, left-to-right thing, mapping to the "q"'s and "r"'s. But if you want to reuse a register allocated with a "q" or "r", you use "0", "1", "2"... etc.

You don't need to put a GCC-allocated register on the clobberlist as GCC knows that you're messing with it.

Now for output registers.

```
asm ("leal (%1,%1,4), %0"
     : "=r" (x_times_5)
     : "r" (x) );
```

Note the use of = to specify an output register. You just have to do it that way. If you want 1 variable to stay in 1 register for both in and out, you have to respecify the register allocated to it on the way in with the "0" type codes as mentioned above.

```
asm ("leal (%0,%0,4), %0"
     : "=r" (x)
     : "0" (x) );
```

This also works, by the way:

```
asm ("leal (%%ebx,%%ebx,4), %%ebx"
     : "=b" (x)
     : "b" (x) );
```

2 things here:

- Note that we don't have to put ebx on the clobberlist, GCC knows it goes into x. Therefore, since it can know the value of ebx, it isn't considered clobbered.
- Notice that in extended asm, you must prefix registers with %% instead of just %. Why, you ask? Because as GCC parses along for %0's and %1's and so on, it would interpret %edx as a %e parameter, see that that's non-existent, and ignore it. Then it would bitch about finding a symbol named dx, which isn't valid because it's not prefixed with % and it's not the one you meant anyway.

Important note: If your assembly statement must execute where you put it, (i.e. must not be moved out of a loop as an optimization), put the keyword volatile after asm and before the ()'s. To be ultra-careful, use

```
__asm__ __volatile__ (...whatever...);
```

However, I would like to point out that if your assembly's only purpose is to calculate the output registers, with no other side effects, you should leave off the volatile keyword so your statement will be processed into GCC's common subexpression elimination optimization.

## Some useful examples

```
#define disable() __asm__ __volatile__ ("cli");

#define enable() __asm__ __volatile__ ("sti");
```

Of course, libc has these defined too.

```
#define times3(arg1, arg2) \
__asm__ ( \
  "leal (%0,%0,2),%0" \
  : "=r" (arg2) \
  : "0" (arg1) );

#define times5(arg1, arg2) \
__asm__ ( \
  "leal (%0,%0,4),%0" \
  : "=r" (arg2) \
  : "0" (arg1) );

#define times9(arg1, arg2) \
__asm__ ( \
  "leal (%0,%0,8),%0" \
  : "=r" (arg2) \
  : "0" (arg1) );
```

These multiply arg1 by 3, 5, or 9 and put them in arg2. You should be ok to do:

```
times5(x,x);
```

as well.

```
#define rep_movsl(src, dest, numwords) \
__asm__ __volatile__ ( \
  "cld\n\t" \
  "rep\n\t" \
  "movsl" \
  : : "S" (src), "D" (dest), "c" (numwords) \
  : "%ecx", "%esi", "%edi" )
```

Helpful Hint: If you say memcpy() with a constant length parameter, GCC will inline it to a rep movsl like above. But if you need a variable length version that inlines and you're always moving dwords, there ya go.

```
#define rep_stosl(value, dest, numwords) \
__asm__ __volatile__ ( \
  "cld\n\t" \
  "rep\n\t" \
  "stosl" \
  : : "a" (value), "D" (dest), "c" (numwords) \
  : "%ecx", "%edi" )
```

Same as above but for memset(), which doesn't get inlined no matter what (for now.)

```
#define RDTSC(llptr) ({ \
__asm__ __volatile__ ( \
        ".byte 0x0f; .byte 0x31" \
        : "=A" (llptr) \
        : : "eax", "edx"); })
```

Reads the TimeStampCounter on the Pentium and puts the 64 bit result into llptr.

## The End

"The End"?! Yah, I guess so.

If you're wondering, I personally am a big fan of AT&T/UNIX syntax now. (It might have helped that I cut my teeth on SPARC assembly. Of course, that machine actually had a decent number of general registers.) It might seem weird to you at first, but it's really more logical than Intel format, and has no ambiguities.

If I still haven't answered a question of yours, look in the Info pages for more information, particularly on the input/output registers. You can do some funky stuff like use "A" to allocate two registers at once for 64-bit math or "m" for static memory locations, and a bunch more that aren't really used as much as "q" and "r".

Alternately, mail me, and I'll see what I can do. (If you find any errors in the above, please, e-mail me and tell me about it! It's frustrating enough to learn without buggy docs!) Or heck, mail me to say "boogabooga."

It's the least you can do.

---

## Related Usenet posts:

- local labels
- fixed point multiplies

---

Thanks to Eric J. Korpela <korpela@ssl.Berkeley.EDU> for some corrections.

---

Have you seen the DJGPP2+Games Page? Probably.
Page written and provided by Brennan Underwood.

Page created with vi, God's own editor.