<https://nullprogram.com/blog/2014/12/09/>

## [How to build DOS COM files with GCC](https://nullprogram.com/blog/2014/12/09/)

December 09, 2014

Update 2018: RenéRebe builds upon this article in an [*interesting follow-up video*](https://www.youtube.com/watch?v=Y7vU5T6rKHE) ([*part 2*](https://www.youtube.com/watch?v=EXiF7g8Hmt4)).

Update 2020: DOS Defender [*was featured on GET OFF MY LAWN*](https://www.youtube.com/watch?v=6UjuFnZYkG4).

This past weekend I participated in [Ludum Dare #31](http://ludumdare.com/compo/2014/12/03/welcome-to-ludum-dare-31/). Before the theme was even announced, due to [recent fascination](https://nullprogram.com/blog/2014/11/22/) I wanted to make an old school DOS game. DOSBox would be the target platform since it’s the most practical way to run DOS applications anymore, despite modern x86 CPUs still being fully backwards compatible all the way back to the 16-bit 8086.

I successfully created and submitted a DOS game called [DOS Defender](http://ludumdare.com/compo/ludum-dare-31/?uid=8472). It’s a 32-bit 80386 real mode DOS COM program. All assets are embedded in the executable and there are no external dependencies, so the entire game is packed into that 10kB binary.

* <https://github.com/skeeto/dosdefender-ld31>
* [DOSDEF.COM](https://github.com/skeeto/dosdefender-ld31/releases/download/1.1.0/DOSDEF.COM) (10kB, v1.1.0, run in DOSBox)



You’ll need a joystick/gamepad in order to play. I included mouse support in the Ludum Dare release in order to make it easier to review, but this was removed because it doesn’t work well.

The most technically interesting part is that **I didn’t need any DOS development tools to create this**! I only used my every day Linux C compiler (gcc). It’s not actually possible to build DOS Defender in DOS. Instead, I’m treating DOS as an embedded platform, which is the only form in which [DOS still exists today](http://www.freedos.org/). Along with DOSBox and [DOSEMU](http://www.dosemu.org/), this is a pretty comfortable toolchain.

If all you care about is how to do this yourself, skip to the “Tricking GCC” section, where we’ll write a “Hello, World” DOS COM program with Linux’s GCC.

### Finding the right tools

I didn’t have GCC in mind when I started this project. What really triggered all of this was that I had noticed Debian’s [bcc](http://linux.die.net/man/1/bcc) package, Bruce’s C Compiler, that builds 16-bit 8086 binaries. It’s kept around for compiling x86 bootloaders and such, but it can also be used to compile DOS COM files, which was the part that interested me.

For some background: the Intel 8086 was a 16-bit microprocessor released in 1978. It had none of the fancy features of today’s CPU: no memory protection, no floating point instructions, and only up to 1MB of RAM addressable. All modern x86 desktops and laptops can still pretend to be a 40-year-old 16-bit 8086 microprocessor, with the same limited addressing and all. That’s some serious backwards compatibility. This feature is called real mode. It’s the mode in which all x86 computers boot. Modern operating systems switch to protected mode as soon as possible, which provides virtual addressing and safe multi-tasking. DOS is not one of these operating systems.

Unfortunately, bcc is not an ANSI C compiler. It supports a subset of K&R C, along with inline x86 assembly. Unlike other 8086 C compilers, it has no notion of “far” or “long” pointers, so inline assembly is required to access [other memory segments](http://en.wikipedia.org/wiki/X86_memory_segmentation) (VGA, clock, etc.). Side note: the remnants of these 8086 “long pointers” still exists today in the Win32 API: LPSTR, LPWORD, LPDWORD, etc. The inline assembly isn’t anywhere near as nice as GCC’s inline assembly. The assembly code has to manually load variables from the stack so, since bcc supports two different calling conventions, the assembly ends up being hard-coded to one calling convention or the other.

Given all its limitations, I went looking for alternatives.

### DJGPP

[DJGPP](http://www.delorie.com/djgpp/) is the DOS port of GCC. It’s a very impressive project, bringing almost all of POSIX to DOS. The DOS ports of many programs are built with DJGPP. In order to achieve this, it only produces 32-bit protected mode programs. If a protected mode program needs to manipulate hardware (i.e. VGA), it must make requests to a [DOS Protected Mode Interface](http://en.wikipedia.org/wiki/DOS_Protected_Mode_Interface) (DPMI) service. If I used DJGPP, I couldn’t make a single, standalone binary as I had wanted, since I’d need to include a DPMI server. There’s also a performance penalty for making DPMI requests.

Getting a DJGPP toolchain working can be difficult, to put it kindly. Fortunately I found a useful project, [build-djgpp](https://github.com/andrewwutw/build-djgpp), that makes it easy, at least on Linux.

Either there’s a serious bug or the official DJGPP binaries [have become infected again](http://www.delorie.com/djgpp/v2faq/faq6_7.html), because in my testing I kept getting the “Not COFF: check for viruses” error message when running my programs in DOSBox. To double check that it’s not an infection on my own machine, I set up a DJGPP toolchain on my Raspberry Pi, to act as a clean room. It’s impossible for this ARM-based device to get infected with an x86 virus. It still had the same problem, and all the binary hashes matched up between the machines, so it’s not my fault.

So given the DPMI issue and the above, I moved on.

### Tricking GCC

What I finally settled on is a neat hack that involves “tricking” GCC into producing real mode DOS COM files, so long as it can target 80386 (as is usually the case). The 80386 was released in 1985 and was the first 32-bit x86 microprocessor. GCC still targets this instruction set today, even in the x86-64 toolchain. Unfortunately, GCC cannot actually produce 16-bit code, so my main goal of targeting 8086 would not be achievable. This doesn’t matter, though, since DOSBox, my intended platform, is an 80386 emulator.

In theory this should even work unchanged with MinGW, but there’s a long-standing MinGW bug that prevents it from working right (“cannot perform PE operations on non PE output file”). It’s still do-able, and I did it myself, but you’ll need to drop the OUTPUT\_FORMAT directive and add an extra objcopy step (objcopy -O binary).

#### Hello World in DOS

To demonstrate how to do all this, let’s make a DOS “Hello, World” COM program using GCC on Linux.

There’s a significant burden with this technique: **there will be no standard library**. It’s basically like writing an operating system from scratch, except for the few services DOS provides. This means no printf() or anything of the sort. Instead we’ll ask DOS to print a string to the terminal. Making a request to DOS means firing an interrupt, which means inline assembly!

DOS has nine interrupts: 0x20, 0x21, 0x22, 0x23, 0x24, 0x25, 0x26, 0x27, 0x2F. The big one, and the one we’re interested in, is 0x21, function 0x09 (print string). Between DOS and BIOS, there are [thousands of functions called this way](http://www.o3one.org/hwdocs/bios_doc/dosref22.html). I’m not going to try to explain x86 assembly, but in short the function number is stuffed into register ah and interrupt 0x21 is fired. Function 0x09 also takes an argument, the pointer to the string to be printed, which is passed in registers dx and ds.

Here’s the GCC inline assembly print() function. Strings passed to this function must be terminated with a $. Why? Because DOS.

static void print(char \*string)

{

asm volatile ("mov $0x09, %%ah\n"

"int $0x21\n"

: /\* no output \*/

: "d"(string)

: "ah");

}

The assembly is declared volatile because it has a side effect (printing the string). To GCC, the assembly is an opaque hunk, and the optimizer relies in the output/input/clobber constraints (the last three lines). For DOS programs like this, all inline assembly will have side effects. This is because it’s not being written for optimization but to access hardware and DOS, things not accessible to plain C.

Care must also be taken by the caller, because GCC doesn’t know that the memory pointed to by string is ever read. It’s likely the array that backs the string needs to be declared volatile too. This is all foreshadowing into what’s to come: doing anything in this environment is an endless struggle against the optimizer. Not all of these battles can be won.

Now for the main function. The name of this function shouldn’t matter, but I’m avoiding calling it main() since MinGW has a funny ideas about mangling this particular symbol, even when it’s asked not to.

int dosmain(void)

{

print("Hello, World!\n$");

return 0;

}

COM files are limited to 65,279 bytes in size. This is because an x86 memory segment is 64kB and COM files are simply loaded by DOS to 0x0100 in the segment and executed. There are no headers, it’s just a raw binary. Since a COM program can never be of any significant size, and no real linking needs to occur (freestanding), the entire thing will be compiled as one translation unit. It will be one call to GCC with a bunch of options.

#### Compiler Options

Here are the essential compiler options.

-std=gnu99 -Os -nostdlib -m32 -march=i386 -ffreestanding

Since no standard libraries are in use, the only difference between gnu99 and c99 is that trigraphs are disabled (as they should be) and inline assembly can be written as asm instead of \_\_asm\_\_. It’s a no brainer. This project will be so closely tied to GCC that I don’t care about using GCC extensions anyway.

I’m using -Os to keep the compiled output as small as possible. It will also make the program run faster. This is important when targeting DOSBox because, by default, it will deliberately run as slow as a machine from the 1980’s. I want to be able to fit in that constraint. If the optimizer is causing problems, you may need to temporarily make this -O0 to determine if the problem is your fault or the optimizer’s fault.

You see, the optimizer doesn’t understand that the program will be running in real mode, and under its addressing constraints. **It will perform all sorts of invalid optimizations that break your perfectly valid programs.** It’s not a GCC bug since we’re doing crazy stuff here. I had to rework my code a number of times to stop the optimizer from breaking my program. For example, I had to avoid returning complex structs from functions because they’d sometimes be filled with garbage. The real danger here is that a future version of GCC will be more clever and will break more stuff. In this battle, volatile is your friend.

Th next option is -nostdlib, since there are no valid libraries for us to link against, even statically.

The options -m32 -march=i386 set the compiler to produce 80386 code. If I was writing a bootloader for a modern computer, targeting 80686 would be fine, too, but DOSBox is 80386.

The -ffreestanding argument requires that GCC not emit code that calls built-in standard library helper functions. Sometimes instead of emitting code to do something, it emits code that calls a built-in function to do it, especially with math operators. This was one of the main problems I had with bcc, where this behavior couldn’t be disabled. This is most commonly used in writing bootloaders and kernels. And now DOS COM files.

#### Linker Options

The -Wl option is used to pass arguments to the linker (ld). We need it since we’re doing all this in one call to GCC.

-Wl,--nmagic,--script=com.ld

The --nmagic turns off page alignment of sections. One, we don’t need this. Two, that would waste precious space. In my tests it doesn’t appear to be necessary, but I’m including it just in case.

The --script option tells the linker that we want to use a custom [linker script](http://wiki.osdev.org/Linker_Scripts). This allows us to precisely lay out the sections (text, data, bss, rodata) of our program. Here’s the com.ld script.

OUTPUT\_FORMAT(binary)

SECTIONS

{

. = 0x0100;

.text :

{

\*(.text);

}

.data :

{

\*(.data);

\*(.bss);

\*(.rodata);

}

\_heap = ALIGN(4);

}

The OUTPUT\_FORMAT(binary) says not to put this into an ELF (or PE, etc.) file. The linker should just dump the raw code. A COM file is just raw code, so this means the linker will produce a COM file!

I had said that COM files are loaded to 0x0100. The fourth line offsets the binary to this location. The first byte of the COM file will still be the first byte of code, but it will be designed to run from that offset in memory.

What follows is all the sections, text (program), data (static data), bss (zero-initialized data), rodata (strings). Finally I mark the end of the binary with the symbol \_heap. This will come in handy later for writing sbrk(), after we’re done with “Hello, World.” I’ve asked for the \_heap position to be 4-byte aligned.

We’re almost there.

#### Program Startup

The linker is usually aware of our entry point (main) and sets that up for us. But since we asked for “binary” output, we’re on our own. If the print() function is emitted first, our program’s execution will begin with executing that function, which is invalid. Our program needs a little header stanza to get things started.

The linker script has a STARTUP option for handling this, but to keep it simple we’ll put that right in the program. This is usually called crt0.o or Boot.o, in case those names every come up in your own reading. This inline assembly must be the very first thing in our code, before any includes and such. DOS will do most of the setup for us, we really just have to jump to the entry point.

asm (".code16gcc\n"

"call dosmain\n"

"mov $0x4C, %ah\n"

"int $0x21\n");

The .code16gcc tells the assembler that we’re going to be running in real mode, so that it makes the proper adjustment. Despite the name, this will not make it produce 16-bit code! First it calls dosmain, the function we wrote above. Then it informs DOS, using function 0x4C (terminate with return code), that we’re done, passing the exit code along in the 1-byte register al (already set by dosmain). This inline assembly is automatically volatile because it has no inputs or outputs.

#### Everything at Once

Here’s the entire C program.

asm (".code16gcc\n"

"call dosmain\n"

"mov $0x4C,%ah\n"

"int $0x21\n");

static void print(char \*string)

{

asm volatile ("mov $0x09, %%ah\n"

"int $0x21\n"

: /\* no output \*/

: "d"(string)

: "ah");

}

int dosmain(void)

{

print("Hello, World!\n$");

return 0;

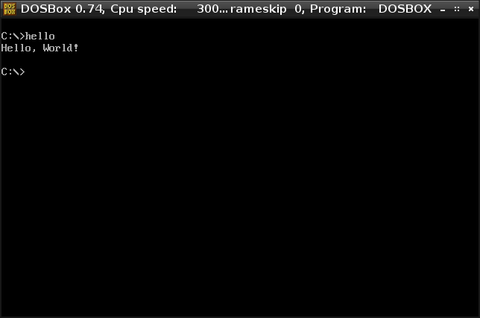
}

I won’t repeat com.ld. Here’s the call to GCC.

gcc -std=gnu99 -Os -nostdlib -m32 -march=i386 -ffreestanding \

-o hello.com -Wl,--nmagic,--script=com.ld hello.c

And testing it in DOSBox:



From here if you want fancy graphics, it’s just a matter of making an interrupt and [writing to VGA memory](http://www.brackeen.com/vga/index.html). If you want sound you can perform an interrupt for the PC speaker. I haven’t sorted out how to call Sound Blaster yet. It was from this point that I grew DOS Defender.

### Memory Allocation

To cover one more thing, remember that \_heap symbol? We can use it to implement sbrk() for dynamic memory allocation within the main program segment. This is real mode, and there’s no virtual memory, so we’re free to write to any memory we can address at any time. Some of this is reserved (i.e. low and high memory) for hardware. So using sbrk() specifically isn’t really necessary, but it’s interesting to implement ourselves.

As is normal on x86, your text and segments are at a low address (0x0100 in this case) and the stack is at a high address (around 0xffff in this case). On Unix-like systems, the memory returned by malloc() comes from two places: sbrk() and mmap(). What sbrk() does is allocates memory just above the text/data segments, growing “up” towards the stack. Each call to sbrk() will grow this space (or leave it exactly the same). That memory would then managed by malloc() and friends.

Here’s how we can get sbrk() in a COM program. Notice I have to define my own size\_t, since we don’t have a standard library.

typedef unsigned short size\_t;

extern char \_heap;

static char \*hbreak = &\_heap;

static void \*sbrk(size\_t size)

{

char \*ptr = hbreak;

hbreak += size;

return ptr;

}

It just sets a pointer to \_heap and grows it as needed. A slightly smarter sbrk() would be careful about alignment as well.

In the making of DOS Defender an interesting thing happened. I was (incorrectly) counting on the memory return by my sbrk() being zeroed. This was the case the first time the game ran. However, DOS doesn’t zero this memory between programs. When I would run my game again, it would pick right up where it left off, because the same data structures with the same contents were loaded back into place. A pretty cool accident! It’s part of what makes this a fun embedded platform.

* [c](https://nullprogram.com/tags/c/)
* [debian](https://nullprogram.com/tags/debian/)
* [tutorial](https://nullprogram.com/tags/tutorial/)
* [game](https://nullprogram.com/tags/game/)

Have a comment on this article? Start a discussion in my [public inbox](https://lists.sr.ht/~skeeto/public-inbox) by sending an email to [~skeeto/public-inbox@lists.sr.ht](mailto:~skeeto/public-inbox@lists.sr.ht?Subject=Re%3A%20How%20to%20build%20DOS%20COM%20files%20with%20GCC) [[mailing list etiquette](https://man.sr.ht/lists.sr.ht/etiquette.md)] , or see [existing discussions](https://lists.sr.ht/~skeeto/public-inbox?search=How+to+build+DOS+COM+files+with+GCC).

This post has [archived comments](https://nullprogram.com/blog/comments/#2014-12-09).

« [LZSS Quine Puzzle](https://nullprogram.com/blog/2014/11/22/)

» [Interactive Programming in C](https://nullprogram.com/blog/2014/12/23/)