Linux Assembly HOWTO

Konstantin Boldyshev

Linux Assembly

konst@linuxassembly.org

Francois−Rene Rideau

Tunes project

fare@tunes.org

This is the Linux Assembly HOWTO, version 0.6d. This document describes how to program in assembly language using *free* programming tools, focusing on development for or from the Linux Operating System, mostly on IA−32 (i386) platform. Included material may or may not be applicable to other hardware and/or software platforms.

Table of Contents

Table of Contents

Chapter 1. Introduction

> **Note:** You can skip this chapter if you are familiar with HOWTOs, or just hate to read all this assembly−unrelated crap.

1.1. Legal Blurb

The most recent official version of this document is available from the Linux Assembly and LDP sites. If you are reading a few−months−old copy, consider checking the above URLs for a new version.

1.2. Foreword

This document aims answering questions of those who program or want to program 32−bit x86 assembly using *free software*, particularly under the Linux operating system. At many places Universal Resource Locators (URL) are given for some software or documentation repository. This document also points to other documents about non−free, non−x86, or non−32−bit assemblers, although this is not its primary goal. Also note that there are FAQs and docs about programming on your favorite platform (whatever it is), which you should consult for platform−specific issues, not related directly to assembly programming.

Because the main interest of assembly programming is to build the guts of operating systems, interpreters, compilers, and games, where C compiler fails to provide the needed expressiveness (performance is more and more seldom as issue), we are focusing on development of such kind of software.

If you don't know what *free software* is, please do read *carefully* the GNU General Public License (GPL or copyleft), which is used in a lot of free software, and is the model for most of their licenses. It generally comes in a file named `COPYING` (or `COPYING.LIB`). Literature from the Free Software Foundation (FSF) might help you too. Particularly, the interesting feature of free software is that it comes with source code which you can consult and correct, or sometimes even borrow from. Read your particular license carefully and do comply to it.

1.3. Contributions

This is an interactively evolving document: you are especially invited to ask questions, to answer questions, to correct given answers, to give pointers to new software, to point the current maintainer to bugs or deficiencies in the pages. In one word, contribute!

To contribute, please contact the maintainer.

> **Note:** At the time of writing, it is Konstantin Boldyshev and no more Francois−Rene Rideau (since version 0.5). I (Fare) had been looking for some time for a serious hacker to replace me as maintainer of this document, and am pleased to announce Konstantin as my worthy successor.

Chapter 2. Do you need assembly?

Well, I wouldn't want to interfere with what you're doing, but here is some advice from the hard−earned experience.

2.1. Pros and Cons

2.1.1. The advantages of Assembly

Assembly can express very low−level things:

- you can access machine−dependent registers and I/O
- you can control the exact code behavior in critical sections that might otherwise involve deadlock between multiple software threads or hardware devices
- you can break the conventions of your usual compiler, which might allow some optimizations (like temporarily breaking rules about memory allocation, threading, calling conventions, etc)
- you can build interfaces between code fragments using incompatible conventions (e.g. produced by different compilers, or separated by a low−level interface)
- you can get access to unusual programming modes of your processor (e.g. 16 bit mode to interface startup, firmware, or legacy code on Intel PCs)
- you can produce reasonably fast code for tight loops to cope with a bad non−optimizing compiler (but then, there are free optimizing compilers available!)
- you can produce hand−optimized code perfectly tuned for your particular hardware setup, though not to anyone else's
- you can write some code for your new language's optimizing compiler (that is something what very few ones will ever do, and even they not often)

2.1.2. The disadvantages of Assembly

Assembly is a very low−level language (the lowest above hand−coding the binary instruction patterns). This means

- it is long and tedious to write initially
- it is quite bug−prone
- your bugs can be very difficult to chase
- your code can be fairly difficult to understand and modify, i.e. to maintain
- the result is non−portable to other architectures, existing or upcoming
- your code will be optimized only for a certain implementation of a same architecture: for instance, among Intel−compatible platforms each CPU design and its variations (relative latency, through−output, and capacity, of processing units, caches, RAM, bus, disks, presence of FPU, MMX, 3DNOW, SIMD extensions, etc) implies potentially completely different optimization techniques. CPU designs already include: Intel 386, 486, Pentium, PPro, PII, PIII, PIV; Cyrix 5x86, 6x86, M2; AMD K5, K6 (K6−2, K6−III), K7 (Athlon, Duron). New designs keep popping up, so don't expect either this listing and your code to be up−to−date.
- you spend more time on a few details and can't focus on small and large algorithmic design, that are known to bring the largest part of the speed up (e.g. you might spend some time building very fast list/array manipulation primitives in assembly; only a hash table would have sped up your program

much more; or, in another context, a binary tree; or some high−level structure distributed over a cluster of CPUs)
- a small change in algorithmic design might completely invalidate all your existing assembly code. So that either you're ready (and able) to rewrite it all, or you're tied to a particular algorithmic design
- On code that ain't too far from what's in standard benchmarks, commercial optimizing compilers outperform hand−coded assembly (well, that's less true on the x86 architecture than on RISC architectures, and perhaps less true for widely available/free compilers; anyway, for typical C code, GCC is fairly good);
- And in any case, as says moderator John Levine on comp.compilers,

  "compilers make it a lot easier to use complex data structures,
  and compilers don't get bored halfway through
  and generate reliably pretty good code."

  They will also *correctly* propagate code transformations throughout the whole (huge) program when optimizing code between procedures and module boundaries.

2.1.3. Assessment

All in all, you might find that though using assembly is sometimes needed, and might even be useful in a few cases where it is not, you'll want to:

- minimize the use of assembly code
- encapsulate this code in well−defined interfaces
- have your assembly code automatically generated from patterns expressed in a higher−level language than assembly (e.g. GCC inline assembly macros)
- have automatic tools translate these programs into assembly code
- have this code be optimized if possible
- All of the above, i.e. write (an extension to) an optimizing compiler back−end.

Even in cases when assembly is needed (e.g. OS development), you'll find that not so much of it is, and that the above principles hold.

See the Linux kernel sources concerning this: as little assembly as needed, resulting in a fast, reliable, portable, maintainable OS. Even a successful game like DOOM was almost massively written in C, with a tiny part only being written in assembly for speed up.

2.2. How to NOT use Assembly

2.2.1. General procedure to achieve efficient code

As says Charles Fiterman on comp.compilers about human vs computer−generated assembly code:

> The human should always win and here is why.
>
> First the human writes the whole thing in a high level language.
> Second he profiles it to find the hot spots where it spends its time.

Third he has the compiler produce assembly for those small sections of code.
Fourth he hand tunes them looking for tiny improvements over the machine generated code.

The human wins because he can use the machine.

2.2.2. Languages with optimizing compilers

Languages like ObjectiveCAML, SML, CommonLISP, Scheme, ADA, Pascal, C, C++, among others, all have free optimizing compilers that will optimize the bulk of your programs, and often do better than hand−coded assembly even for tight loops, while allowing you to focus on higher−level details, and without forbidding you to grab a few percent of extra performance in the above−mentioned way, once you've reached a stable design. Of course, there are also commercial optimizing compilers for most of these languages, too!

Some languages have compilers that produce C code, which can be further optimized by a C compiler: LISP, Scheme, Perl, and many other. Speed is fairly good.

2.2.3. General procedure to speed your code up

As for speeding code up, you should do it only for parts of a program that a profiling tool has consistently identified as being a performance bottleneck.

Hence, if you identify some code portion as being too slow, you should

- first try to use a better algorithm;
- then try to compile it rather than interpret it;
- then try to enable and tweak optimization from your compiler;
- then give the compiler hints about how to optimize (typing information in LISP; register usage with GCC; lots of options in most compilers, etc).
- then possibly fallback to assembly programming

Finally, before you end up writing assembly, you should inspect generated code, to check that the problem really is with bad code generation, as this might really not be the case: compiler−generated code might be better than what you'd have written, particularly on modern multi−pipelined architectures! Slow parts of a program might be intrinsically so. The biggest problems on modern architectures with fast processors are due to delays from memory access, cache−misses, TLB−misses, and page−faults; register optimization becomes useless, and you'll more profitably re−think data structures and threading to achieve better locality in memory access. Perhaps a completely different approach to the problem might help, then.

2.2.4. Inspecting compiler−generated code

There are many reasons to inspect compiler−generated assembly code. Here is what you'll do with such code:

- check whether generated code can be obviously enhanced with hand−coded assembly (or by tweaking compiler switches)
- when that's the case, start from generated code and modify it instead of starting from scratch
- more generally, use generated code as stubs to modify, which at least gets right the way your

assembly routines interface to the external world
- track down bugs in your compiler (hopefully the rarer)

The standard way to have assembly code be generated is to invoke your compiler with the `-S` flag. This works with most Unix compilers, including the GNU C Compiler (GCC), but YMMV. As for GCC, it will produce more understandable assembly code with the `-fverbose-asm` command–line option. Of course, if you want to get good assembly code, don't forget your usual optimization options and hints!

2.3. Linux and assembly

As you probably noticed, in general case you don't need to use assembly language in Linux programming. Unlike DOS, you do not have to write Linux drivers in assembly (well, actually you can do it if you really want). And with modern optimizing compilers, if you care of speed optimization for different CPU's, it's much simpler to write in C. However, if you're reading this, you might have some reason to use assembly instead of C/C++.

You may *need* to use assembly, or you may *want* to use assembly. In short, main practical (*need*) reasons of diving into the assembly realm are *small code* and *libc independence*. Impractical (*want*), and the most often reason is being just an old crazy hacker, who has twenty years old habit of doing everything in assembly language.

However, if you're porting Linux to some embedded hardware you can be quite short at the size of whole system: you need to fit kernel, libc and all that stuff of (file|find|text|sh|etc.) utils into several hundreds of kilobytes, and every kilobyte costs much. So, one of the possible ways is to rewrite some (or all) parts of system in assembly, and this will really save you a lot of space. For instance, a simple **httpd** written in assembly can take less than 600 bytes; you can fit a webserver, consisting of kernel and httpd, in 400 KB or less... Think about it.

Chapter 3. Assemblers

3.1. GCC Inline Assembly

The well−known GNU C/C++ Compiler (GCC), an optimizing 32−bit compiler at the heart of the GNU project, supports the x86 architecture quite well, and includes the ability to insert assembly code in C programs, in such a way that register allocation can be either specified or left to GCC. GCC works on most available platforms, notably Linux, *BSD, VSTa, OS/2, *DOS, Win*, etc.

3.1.1. Where to find GCC

The original GCC site is the GNU FTP site ftp://prep.ai.mit.edu/pub/gnu/gcc/ together with all released application software from the GNU project. Linux−configured and pre−compiled versions can be found in ftp://metalab.unc.edu/pub/Linux/GCC/ There are a lot of FTP mirrors of both sites everywhere around the world, as well as CD−ROM copies.

GCC development has split into two branches some time ago (GCC 2.8 and EGCS), but they merged back, and current GCC webpage is http://gcc.gnu.org.

Sources adapted to your favorite OS and pre−compiled binaries should be found at your usual FTP sites.

DOS port of GCC is called DJGPP.

There are two Win32 GCC ports: cygwin and mingw

There is also an OS/2 port of GCC called EMX; it works under DOS too, and includes lots of unix−emulation library routines. Look around the following site: ftp://ftp−os2.cdrom.com/pub/os2/emx09c.

3.1.2. Where to find docs for GCC Inline Asm

The documentation of GCC includes documentation files in TeXinfo format. You can compile them with TeX and print then result, or convert them to `.info`, and browse them with emacs, or convert them to `.html`, or nearly whatever you like; convert (with the right tools) to whatever you like, or just read as is. The `.info` files are generally found on any good installation for GCC.

The right section to look for is `C Extensions::Extended Asm::`

Section `Invoking GCC::Submodel Options::i386 Options::` might help too. Particularly, it gives the i386 specific constraint names for registers: `abcdSDB` correspond to `%eax`, `%ebx`, `%ecx`, `%edx`, `%esi`, `%edi` and `%ebp` respectively (no letter for `%esp`).

The DJGPP Games resource (not only for game hackers) had page specifically about assembly, but it's down. Its data have nonetheless been recovered on the DJGPP site, that contains a mine of other useful information: http://www.delorie.com/djgpp/doc/brennan/, and in the DJGPP Quick ASM Programming Guide.

GCC depends on GAS for assembling and follows its syntax (see below); do mind that inline asm needs percent characters to be quoted, they will be passed to GAS. See the section about GAS below.

Find *lots* of useful examples in the `linux/include/asm−i386/` subdirectory of the sources for the Linux kernel.

3.1.3. Invoking GCC to build proper inline assembly code

Because assembly routines from the kernel headers (and most likely your own headers, if you try making your assembly programming as clean as it is in the linux kernel) are embedded in `extern inline` functions, GCC must be invoked with the `−O` flag (or `−O2`, `−O3`, etc), for these routines to be available. If not, your code may compile, but not link properly, since it will be looking for non−inlined `extern` functions in the libraries against which your program is being linked! Another way is to link against libraries that include fallback versions of the routines.

Inline assembly can be disabled with `−fno−asm`, which will have the compiler die when using extended inline asm syntax, or else generate calls to an external function named `asm()` that the linker can't resolve. To counter such flag, `−fasm` restores treatment of the `asm` keyword.

More generally, good compile flags for GCC on the x86 platform are

gcc −O2 −fomit−frame−pointer −W −Wall

`−O2` is the good optimization level in most cases. Optimizing besides it takes more time, and yields code that is much larger, but only a bit faster; such over−optimization might be useful for tight loops only (if any), which you may be doing in assembly anyway. In cases when you need really strong compiler optimization for a few files, do consider using up to `−O6`.

`−fomit−frame−pointer` allows generated code to skip the stupid frame pointer maintenance, which makes code smaller and faster, and frees a register for further optimizations. It precludes the easy use of debugging tools (**gdb**), but when you use these, you just don't care about size and speed anymore anyway.

`−W −Wall` enables all useful warnings and helps you to catch obvious stupid errors.

You can add some CPU−specific `−m486` or such flag so that GCC will produce code that is more adapted to your precise CPU. Note that modern GCC has `−mpentium` and such flags (and PGCC has even more), whereas GCC 2.7.x and older versions do not. A good choice of CPU−specific flags should be in the Linux kernel. Check the TeXinfo documentation of your current GCC installation for more.

`−m386` will help optimize for size, hence also for speed on computers whose memory is tight and/or loaded, since big programs cause swap, which more than counters any "optimization" intended by the larger code. In such settings, it might be useful to stop using C, and use instead a language that favors code factorization, such as a functional language and/or FORTH, and use a bytecode− or wordcode− based implementation.

Note that you can vary code generation flags from file to file, so performance−critical files will use maximum optimization, whereas other files will be optimized for size.

To optimize even more, option `−mregparm=2` and/or corresponding function attribute might help, but might pose lots of problems when linking to foreign code, *including libc*. There are ways to correctly declare foreign functions so the right call sequences be generated, or you might want to recompile the foreign libraries to use the same register−based calling convention...

Note that you can add make these flags the default by editing file `/usr/lib/gcc-lib/i486-linux/2.7.2.3/specs` or wherever that is on your system (better not add `-W -Wall` there, though). The exact location of the GCC specs files on system can be found by **gcc -v**.

3.1.4. Macro support

GCC allows (and requires) you to specify register constraints in your inline assembly code, so the optimizer always know about it; thus, inline assembly code is really made of patterns, not forcibly exact code.

Thus, you can make put your assembly into CPP macros, and inline C functions, so anyone can use it in as any C function/macro. Inline functions resemble macros very much, but are sometimes cleaner to use. Beware that in all those cases, code will be duplicated, so only local labels (of `1:` style) should be defined in that asm code. However, a macro would allow the name for a non local defined label to be passed as a parameter (or else, you should use additional meta−programming methods). Also, note that propagating inline asm code will spread potential bugs in them; so watch out doubly for register constraints in such inline asm code.

Lastly, the C language itself may be considered as a good abstraction to assembly programming, which relieves you from most of the trouble of assembling.

3.2. GAS

GAS is the GNU Assembler, that GCC relies upon.

3.2.1. Where to find it

Find it at the same place where you found GCC, in a package named binutils.

The latest version is available from HJLu at ftp://ftp.varesearch.com/pub/support/hjl/binutils/.

3.2.2. What is this AT&T syntax

Because GAS was invented to support a 32−bit unix compiler, it uses standard AT&T syntax, which resembles a lot the syntax for standard m68k assemblers, and is standard in the UNIX world. This syntax is neither worse, nor better than the Intel syntax. It's just different. When you get used to it, you find it much more regular than the Intel syntax, though a bit boring.

Here are the major caveats about GAS syntax:

- Register names are prefixed with `%`, so that registers are `%eax`, `%dl` and so on, instead of just `eax`, `dl`, etc. This makes it possible to include external C symbols directly in assembly source, without any risk of confusion, or any need for ugly underscore prefixes.
- The order of operands is source(s) first, and destination last, as opposed to the Intel convention of destination first and sources last. Hence, what in Intel syntax is `mov eax,edx` (move contents of register `edx` into register `eax`) will be in GAS syntax `mov %edx,%eax`.

- The operand size is specified as a suffix to the instruction name. The suffix is `b` for (8−bit) byte, `w` for (16−bit) word, and `l` for (32−bit) long. For instance, the correct syntax for the above instruction would have been `movl %edx,%eax`. However, gas does not require strict AT&T syntax, so the suffix is optional when size can be guessed from register operands, and else defaults to 32−bit (with a warning).
- Immediate operands are marked with a `$` prefix, as in `addl $5,%eax` (add immediate long value 5 to register `%eax`).
- Missing operand prefix indicates it is a memory−address; hence `movl $foo,%eax` puts the *address* of variable `foo` into register `%eax`, but `movl foo,%eax` puts the *contents* of variable `foo` into register `%eax`.
- Indexing or indirection is done by enclosing the index register or indirection memory cell address in parentheses, as in `testb $0x80,17(%ebp)` (test the high bit of the byte value at offset 17 from the cell pointed to by `%ebp`).

Note: There are few programs which may help you to convert source code between AT&T and Intel assembler syntaxes; some of the are capable of performing conversion in both directions.

GAS has comprehensive documentation in TeXinfo format, which comes at least with the source distribution. Browse extracted `.info` pages with Emacs or whatever. There used to be a file named gas.doc or as.doc around the GAS source package, but it was merged into the TeXinfo docs. Of course, in case of doubt, the ultimate documentation is the sources themselves! A section that will particularly interest you is `Machine Dependencies::i386-Dependent::`

Again, the sources for Linux (the OS kernel) come in as excellent examples; see under `linux/arch/i386/` the following files: `kernel/*.S`, `boot/compressed/*.S`, `mathemu/*.S`.

If you are writing kind of a language, a thread package, etc., you might as well see how other languages (OCaml, Gforth, etc.), or thread packages (QuickThreads, MIT pthreads, LinuxThreads, etc), or whatever else do it.

Finally, just compiling a C program to assembly might show you the syntax for the kind of instructions you want. See section Do you need assembly? above.

3.2.3. Intel syntax

Good news are that starting from binutils 2.10 release, GAS supports Intel syntax too. It can be triggered with `.intel_syntax` directive.

3.2.4. 16−bit mode

Binutils (2.9.1.0.25+) now fully support 16−bit mode (registers *and* addressing) on i386 PCs. Use `.code16` and `.code32` to switch between assembly modes.

Also, a neat trick used by several people (including the oskit authors) is to force GCC to produce code for 16−bit real mode, using an inline assembly statement `asm(".code16\n")`. GCC will still emit only 32−bit addressing modes, but GAS will insert proper 32−bit prefixes for them.

3.2.5. Macro support

GAS has some macro capability included, as detailed in the texinfo docs. Moreover, while GCC recognizes `.s` files as raw assembly to send to GAS, it also recognizes `.S` files as files to pipe through CPP before feeding them to GAS. Again and again, see Linux sources for examples.

GAS also has GASP (GAS Preprocessor), which adds all the usual macroassembly tricks to GAS. GASP comes together with GAS in the GNU binutils archive. It works as a filter, like CPP and M4. I have no idea on details, but it comes with its own texinfo documentation, which you would like to browse (**info gasp**), print, grok. GAS with GASP looks like a regular macro−assembler to me.

3.3. NASM

The Netwide Assembler project provides cool i386 assembler, written in C, that should be modular enough to eventually support all known syntaxes and object formats.

3.3.1. Where to find NASM

http://nasm.2y.net, http://www.cryogen.com/nasm/

Binary release on your usual metalab mirror in `devel/lang/asm/` directory. Should also be available as `.rpm` or `.deb` in your usual RedHat/Debian distributions' contrib.

3.3.2. What it does

The syntax is Intel−style. Comprehensive macroprocessing support is integrated.

Supported object file formats are `bin`, `aout`, `coff`, `elf`, `as86`, `obj` (DOS), `win32`, `rdf` (their own format).

NASM can be used as a backend for the free LCC compiler (support files included).

Unless you're using BCC as a 16−bit compiler (which is out of scope of this 32−bit HOWTO), you should definitely use NASM instead of say AS86 or MASM, because it runs on all platforms.

> **Note:** NASM comes with a disassembler, NDISASM.

Its hand−written parser makes it much faster than GAS, though of course, it doesn't support three bazillion different architectures. If you like Intel−style syntax, as opposed to GAS syntax, then it should be the assembler of choice..

Note: There are few programs which may help you to convert source code between AT&T and Intel assembler syntaxes; some of the are capable of performing conversion in both directions.

3.4. AS86

AS86 is a 80x86 assembler, both 16−bit and 32−bit, with integrated macro support. It has mostly Intel−syntax, though it differs slightly as for addressing modes.

3.4.1. Where to get AS86

Current version is 0.15.4, it can be found at http://www.cix.co.uk/~mayday/, in bin86 package with linker (ld86), or as separate archive.

> **Note:** A completely outdated version 0.4 of AS86 is distributed by HJLu just to compile the Linux kernel versions prior to 2.4, in a package named bin86, available in any Linux GCC repository. But I advise no one to use it for anything else but compiling Linux. This version supports only a hacked minix object file format, which is not supported by the GNU binutils or anything, and it has a few bugs in 32−bit mode, so you really should better keep it only for compiling Linux.

3.4.2. Where to find docs

See the man page and as.doc from the source package. When in doubt, the sources themselves are often a good docs: they aren't very well commented, but the programming style is straightforward. You might try to see how as86 is used in ELKS, LILO, or Tunes 0.0.0.25...

3.4.3. Using AS86 with BCC

Here's the GNU Makefile entry for using BCC to transform `.s` asm into both a.out `.o` object and `.l` listing:

```
%.o %.l:        %.s
        bcc -3 -G -c -A-d -A-l -A$*.l -o $*.o $<
```

Remove the `%.l`, `-A-l`, and `-A$*.l`, if you don't want any listing. If you want something else than a.out, you can examine BCC docs about the other supported formats, and/or use the objcopy utility from the GNU binutils package.

3.5. Other Assemblers

These are other non−regular options, in case the previous didn't satisfy you (why?), that I don't recommend in the usual (?) case, but that could be quite useful if the assembler must be integrated in the software you're designing (i.e. an OS or development environment).

3.5.1. Free Pascal

Free Pascal has an internal 32−bit assembler (based on NASM tables) and a switchable output that allows:

- Binary (ELF and coff when crosscompiled .o) output
- NASM
- MASM
- TASM
- AS (aout,coff, elf32)

The MASM and TASM output are not as good debugged as the other two, but can be handy sometimes.

The assembler's look and feel are based on Turbo Pascal's internal BASM, and the IDE supports similar highlighting, and FPC can fully integrate with gcc (on C level, not C++).

Using a dummy RTL, one can even generate pure assembler programs.

3.5.2. Win32Forth assembler

Win32Forth is a *free* 32−bit ANS FORTH system that successfully runs under Win32s, Win95, Win/NT. It includes a free 32−bit assembler (either prefix or postfix syntax) integrated into the reflective FORTH language. Macro processing is done with the full power of the reflective language FORTH; however, the only supported input and output contexts is Win32For itself (no dumping of `.obj` file, but you could add that feature yourself, of course). Find it at ftp://ftp.forth.org/pub/Forth/Compilers/native/windows/Win32For/.

3.5.3. SHASM

SHASM is an assembler written in GNU Bash Version 2, which may work in other recent unix−style "shell" command interpreters. It is a trivially extensible, utterly flexible collection of unix shell routines for assembling arbitrary binary files, including 80386 machine language programs. SHASM uses echo −e \000, (implicit) "let"−style expressions including bitwise Booleans, arrays, and perhaps other non−Bourne features of Bash. SHASM does NOT call externals such as sed, dd, expr and so on. All it needs is the shell, although it does need a cushy shell.

It is (of course) slower than other assemblers. It has its own syntax (well, as most of cLIeNUX does :). Fairly good documentation is included. Check it out: ftp://linux01.gwdg.de/pub/cLIeNUX/interim/shasm.TGZ. Probably you'll not use it, but at least it deserves your interest as a crazy idea.

3.5.4. TDASM

The Table Driven Assembler (TDASM) is a *free* portable cross assembler for any kind of assembly language. It should be possible to use it as a compiler to any target microprocessor using a table that defines the compilation process.

It is available from http://www.penguin.cz/~niki/tdasm/.

3.5.5. Terse

Terse is a programming tool that provides *THE* most compact assembler syntax for the x86 family! However, it is evil proprietary software. It is said that there was a project for a free clone somewhere, that was abandoned after worthless pretenses that the syntax would be owned by the original author. Thus, if you're looking for a nifty programming project related to assembly hacking, I invite you to develop a terse−syntax frontend to NASM, if you like that syntax.

As an interesting historic remark, on comp.compilers,

1999/07/11 19:36:51, the moderator wrote:

"There's no reason that assemblers have to have awful syntax. About
30 years ago I used Niklaus Wirth's PL360, which was basically a S/360
assembler with Algol syntax and a a little syntactic sugar like while
loops that turned into the obvious branches. It really was an
assembler, e.g., you had to write out your expressions with explicit
assignments of values to registers, but it was nice. Wirth used it to
write Algol W, a small fast Algol subset, which was a predecessor to
Pascal. As is so often the case, Algol W was a significant
improvement over many of its successors. −John"

3.5.6. HLA

HLA is a *H*igh *L*evel *A*ssembly language. It uses a high level language like syntax (similar to Pascal, C/C++, and other HLLs) for variable declarations, procedure declarations, and procedure calls. It uses a modified assembly language syntax for the standard machine instructions. It also provides several high level language style control structures (if, while, repeat..until, etc.) that help you write much more readable code.

HLA is free, but runs only under Win32. You need MASM and a 32−bit version of MS−link, because HLA produces MASM code and uses MASM for final assembling and linking. However it comes with **m2t** (MASM to TASM) post−processor program that converts the HLA MASM output to a form that will compile under TASM. Unfortunately, NASM is not supported.

3.5.7. TALC

TALC is another free MASM/Win32 based compiler (however it supports ELF output, does it?).

TAL stands for *T*yped *A*ssembly *L*anguage. It extends traditional untyped assembly languages with typing annotations, memory management primitives, and a sound set of typing rules, to guarantee the memory safety, control flow safety,and type safety of TAL programs. Moreover, the typing constructs are expressive enough to encode most source language programming features including records and structures, arrays, higher−order and polymorphic functions, exceptions, abstract data types, subtyping, and modules. Just as importantly, TAL is flexible enough to admit many low−level compiler optimizations. Consequently, TAL is an ideal target platform for type−directed compilers that want to produce verifiably safe code for use in secure mobile code applications or extensible operating system kernels.

3.5.8. Non−free and/or Non−32bit x86 assemblers

You may find more about them, together with the basics of x86 assembly programming, in the Raymond Moon's x86 assembly FAQ.

Note that all DOS−based assemblers should work inside the Linux DOS Emulator, as well as other similar emulators, so that if you already own one, you can still use it inside a real OS. Recent DOS−based assemblers also support COFF and/or other object file formats that are supported by the GNU BFD library, so that you can use them together with your free 32−bit tools, perhaps using GNU objcopy (part of the binutils) as a conversion filter.

Chapter 4. Metaprogramming

Assembly programming is a bore, but for critical parts of programs.

You should use the appropriate tool for the right task, so don't choose assembly when it does not fit; C, OCaml, perl, Scheme, might be a better choice in the most cases.

However, there are cases when these tools do not give fine enough control on the machine, and assembly is useful or needed. In these cases you'll appreciate a system of macroprocessing and metaprogramming that allows recurring patterns to be factored each into one indefinitely reusable definition, which allows safer programming, automatic propagation of pattern modification, etc. Plain assembler often is not enough, even when one is doing only small routines to link with C.

4.1. External filters

Whatever is the macro support from your assembler, or whatever language you use (even C!), if the language is not expressive enough to you, you can have files passed through an external filter with a Makefile rule like that:

```
%.s:    %.S other_dependencies
        $(FILTER) $(FILTER_OPTIONS) < $< > $@
```

4.1.1. CPP

CPP is truly not very expressive, but it's enough for easy things, it's standard, and called transparently by GCC.

As an example of its limitations, you can't declare objects so that destructors are automatically called at the end of the declaring block; you don't have diversions or scoping, etc.

CPP comes with any C compiler. However, considering how mediocre it is, stay away from it if by chance you can make it without C.

4.1.2. M4

M4 gives you the full power of macroprocessing, with a Turing equivalent language, recursion, regular expressions, etc. You can do with it everything that CPP cannot.

See macro4th (this4th) or the Tunes 0.0.0.25 sources as examples of advanced macroprogramming using m4.

However, its disfunctional quoting and unquoting semantics force you to use explicit continuation−passing tail−recursive macro style if you want to do *advanced* macro programming (which is remindful of TeX −− BTW, has anyone tried to use TeX as a macroprocessor for anything else than typesetting ?). This is NOT worse than CPP that does not allow quoting and recursion anyway.

The right version of M4 to get is `GNU m4 1.4` (or later if exists), which has the most features and the least bugs or limitations of all. m4 is designed to be slow for anything but the simplest uses, which might still be ok for most assembly programming (you are not writing million−lines assembly programs, are you?).

4.1.3. Macroprocessing with your own filter

You can write your own simple macro−expansion filter with the usual tools: perl, awk, sed, etc. It can be made rather quickly, and you control everything. But, of course, power in macroprocessing implies "the hard way".

4.2. Metaprogramming

Instead of using an external filter that expands macros, one way to do things is to write programs that write part or all of other programs.

For instance, you could use a program outputting source code

- to generate sine/cosine/whatever lookup tables,
- to extract a source−form representation of a binary file,
- to compile your bitmaps into fast display routines,
- to extract documentation, initialization/finalization code, description tables, as well as normal code from the same source files,
- to have customized assembly code, generated from a perl/shell/scheme script that does arbitrary processing,
- to propagate data defined at one point only into several cross−referencing tables and code chunks.
- etc.

Think about it!

4.2.1. Backends from compilers

Compilers like GCC, SML/NJ, Objective CAML, MIT−Scheme, CMUCL, etc, do have their own generic assembler backend, which you might choose to use, if you intend to generate code semi−automatically from the according languages, or from a language you hack: rather than write great assembly code, you may instead modify a compiler so that it dumps great assembly code!

4.2.2. The New−Jersey Machine−Code Toolkit

There is a project, using the programming language Icon (with an experimental ML version), to build a basis for producing assembly−manipulating code. See around http://www.eecs.harvard.edu/~nr/toolkit/

4.2.3. TUNES

The TUNES Project for a Free Reflective Computing System is developing its own assembler as an extension to the Scheme language, as part of its development process. It doesn't run at all yet, though help is welcome.

The assembler manipulates abstract syntax trees, so it could equally serve as the basis for a assembly syntax translator, a disassembler, a common assembler/compiler back−end, etc. Also, the full power of a real language, Scheme, make it unchallenged as for macroprocessing/metaprogramming.

Chapter 5. Calling conventions

5.1. Linux

5.1.1. Linking to GCC

This is the preferred way if you are developing mixed C−asm project. Check GCC docs and examples from Linux kernel `.S` files that go through gas (not those that go through as86).

32−bit arguments are pushed down stack in reverse syntactic order (hence accessed/popped in the right order), above the 32−bit near return address. `%ebp`, `%esi`, `%edi`, `%ebx` are callee−saved, other registers are caller−saved; `%eax` is to hold the result, or `%edx:%eax` for 64−bit results.

FP stack: I'm not sure, but I think result is in `st(0)`, whole stack caller−saved.

Note that GCC has options to modify the calling conventions by reserving registers, having arguments in registers, not assuming the FPU, etc. Check the i386 `.info` pages.

Beware that you must then declare the `cdecl` or `regparm(0)` attribute for a function that will follow standard GCC calling conventions. See `C Extensions::Extended Asm::` section from the GCC info pages. See also how Linux defines its `asmlinkage` macro...

5.1.2. ELF vs a.out problems

Some C compilers prepend an underscore before every symbol, while others do not.

Particularly, Linux a.out GCC does such prepending, while Linux ELF GCC does not.

If you need to cope with both behaviors at once, see how existing packages do. For instance, get an old Linux source tree, the Elk, qthreads, or OCaml...

You can also override the implicit C−>asm renaming by inserting statements like

```
        void foo asm("bar") (void);
```

to be sure that the C function `foo()` will be called really `bar` in assembly.

Note that the **objcopy** utility from the binutils package should allow you to transform your a.out objects into ELF objects, and perhaps the contrary too, in some cases. More generally, it will do lots of file format conversions.

5.1.3. Direct Linux syscalls

Often you will be told that using C library (libc) is the only way, and direct system calls are bad. This is true. To some extent. In general, you must know that libc is not sacred, and in *most* cases it only does some checks, then calls kernel, and then sets errno. You can easily do this in your program as well (if you need to), and your program will be dozen times smaller, and this will result in improved performance as well, just because you're not using shared libraries (static binaries are faster). Using or not using libc in assembly programming is more a question of taste/belief than something practical. Remember, Linux is aiming to be POSIX compliant, so does libc. This means that syntax of almost all libc "system calls" exactly matches syntax of real kernel system calls (and vice versa). Besides, GNU libc(glibc) becomes slower and slower from version to version, and eats more and more memory; and so, cases of using direct system calls become quite usual. But.. main drawback of throwing libc away is that possibly you will need to implement several libc specific functions (that are not just syscall wrappers) on your own (`printf()` and Co.).. and you are ready for that, aren't you? :)

Here is summary of direct system calls pros and cons.

Pros:

- the smallest possible size; squeezing the last byte out of the system
- the highest possible speed; squeezing cycles out of your favorite benchmark
- full control: you can adapt your program/library to your specific language or memory requirements or whatever
- no pollution by libc cruft
- no pollution by C calling conventions (if you're developing your own language or environment)
- static binaries make you independent from libc upgrades or crashes, or from dangling `#!` path to an interpreter (and are faster)
- just for the fun out of it (don't you get a kick out of assembly programming?)

Cons:

- If any other program on your computer uses the libc, then duplicating the libc code will actually wastes memory, not saves it.
- Services redundantly implemented in many static binaries are a waste of memory. But you can make your libc replacement a shared library.
- Size is much better saved by having some kind of bytecode, wordcode, or structure interpreter than by writing everything in assembly. (the interpreter itself could be written either in C or assembly.) The best way to keep multiple binaries small is to not have multiple binaries, but instead to have an interpreter process files with `#!` prefix. This is how OCaml works when used in wordcode mode (as opposed to optimized native code mode), and it is compatible with using the libc. This is also how Tom Christiansen's Perl PowerTools reimplementation of unix utilities works. Finally, one last way to keep things small, that doesn't depend on an external file with a hardcoded path, be it library or interpreter, is to have only one binary, and have multiply−named hard or soft links to it: the same binary will provide everything you need in an optimal space, with no redundancy of subroutines or useless binary headers; it will dispatch its specific behavior according to its *`argv[0]`*; in case it isn't called with a recognized name, it might default to a shell, and be possibly thus also usable as an interpreter!
- You cannot benefit from the many functionalities that libc provides besides mere linux syscalls: that is, functionality described in section 3 of the manual pages, as opposed to section 2, such as malloc, threads, locale, password, high−level network management, etc.
- Therefore, you might have to reimplement large parts of libc, from `printf()` to `malloc()` and `gethostbyname`. It's redundant with the libc effort, and can be *quite* boring sometimes. Note that some people have already reimplemented "light" replacements for parts of the libc −− check them out! (Redhat's minilibc, Rick Hohensee's libsys, Felix von Leitner's dietlibc, Christian Fowelin's libASM, asmutils project is working on pure assembly libc)
- Static libraries prevent you to benefit from libc upgrades as well as from libc add−ons such as the zlibc package, that does on−the−fly transparent decompression of gzip−compressed files.

- The few instructions added by the libc can be a *ridiculously* small speed overhead as compared to the cost of a system call. If speed is a concern, your main problem is in your usage of system calls, not in their wrapper's implementation.
- Using the standard assembly API for system calls is much slower than using the libc API when running in micro−kernel versions of Linux such as L4Linux, that have their own faster calling convention, and pay high convention−translation overhead when using the standard one (L4Linux comes with libc recompiled with their syscall API; of course, you could recompile your code with their API, too).
- See previous discussion for general speed optimization issue.
- If syscalls are too slow to you, you might want to hack the kernel sources (in C) instead of staying in userland.

If you've pondered the above pros and cons, and still want to use direct syscalls, then here is some advice.

- You can easily define your system calling functions in a portable way in C (as opposed to unportable using assembly), by including `asm/unistd.h`, and using provided macros.
- Since you're trying to replace it, go get the sources for the libc, and grok them. (And if you think you can do better, then send feedback to the authors!)
- As an example of pure assembly code that does everything you want, examine Linux assembly resources.

Basically, you issue an `int 0x80`, with the `__NR_`syscallname number (from `asm/unistd.h`) in `eax`, and parameters (up to six) in `ebx`, `ecx`, `edx`, `esi`, `edi`, `ebp` respectively.

Result is returned in `eax`, with a negative result being an error, whose opposite is what libc would put into `errno`. The user−stack is not touched, so you needn't have a valid one when doing a syscall.

> **Note:** Passing sixth parameter in `ebp` appeared in Linux 2.4, previous Linux versions understand only 5 parameters in registers.

Linux Kernel Internals, and especially How System Calls Are Implemented on i386 Architecture? chapter will give you more robust overview.

As for the invocation arguments passed to a process upon startup, the general principle is that the stack originally contains the number of arguments *`argc`*, then the list of pointers that constitute *`*argv`*, then a null−terminated sequence of null−terminated `variable=value` strings for the *`environ`*ment. For more details, do examine Linux assembly resources, read the sources of C startup code from your libc (`crt0.S` or `crt1.S`), or those from the Linux kernel (`exec.c` and `binfmt_*.c` in `linux/fs/`).

5.1.4. Hardware I/O under Linux

If you want to perform direct port I/O under Linux, either it's something very simple that does not need OS arbitration, and you should see the `IO−Port−Programming` mini−HOWTO; or it needs a kernel device driver, and you should try to learn more about kernel hacking, device driver development, kernel modules, etc, for which there are other excellent HOWTOs and documents from the LDP.

Particularly, if what you want is Graphics programming, then do join one of the GGI or XFree86 projects.

Some people have even done better, writing small and robust XFree86 drivers in an interpreted domain−specific language, GAL, and achieving the efficiency of hand C−written drivers through partial evaluation (drivers not only not in asm, but not even in C!). The problem is that the partial evaluator they used to achieve efficiency is not free software. Any taker for a replacement?

Anyway, in all these cases, you'll be better when using GCC inline assembly with the macros from `linux/asm/*.h` than writing full assembly source files.

5.1.5. Accessing 16−bit drivers from Linux/i386

Such thing is theoretically possible (proof: see how DOSEMU can selectively grant hardware port access to programs), and I've heard rumors that someone somewhere did actually do it (in the PCI driver? Some VESA access stuff? ISA PnP? dunno). If you have some more precise information on that, you'll be most welcome. Anyway, good places to look for more information are the Linux kernel sources, DOSEMU sources (and other programs in the DOSEMU repository), and sources for various low−level programs under Linux... (perhaps GGI if it supports VESA).

Basically, you must either use 16−bit protected mode or vm86 mode.

The first is simpler to setup, but only works with well−behaved code that won't do any kind of segment arithmetics or absolute segment addressing (particularly addressing segment 0), unless by chance it happens that all segments used can be setup in advance in the LDT.

The later allows for more "compatibility" with vanilla 16−bit environments, but requires more complicated handling.

In both cases, before you can jump to 16−bit code, you must

- mmap any absolute address used in the 16−bit code (such as ROM, video buffers, DMA targets, and memory−mapped I/O) from `/dev/mem` to your process' address space,
- setup the LDT and/or vm86 mode monitor.
- grab proper I/O permissions from the kernel (see the above section)

Again, carefully read the source for the stuff contributed to the DOSEMU project, particularly these mini−emulators for running ELKS and/or simple `.COM` programs under Linux/i386.

5.2. DOS and Windows

Most DOS extenders come with some interface to DOS services. Read their docs about that, but often, they just simulate `int 0x21` and such, so you do "as if" you are in real mode (I doubt they have more than stubs and extend things to work with 32−bit operands; they most likely will just reflect the interrupt into the real−mode or vm86 handler).

Docs about DPMI (and much more) can be found on ftp://x2ftp.oulu.fi/pub/msdos/programming/ (again, the original x2ftp site is closing (no more?), so use a mirror site).

DJGPP comes with its own (limited) glibc derivative/subset/replacement, too.

It is possible to cross−compile from Linux to DOS, see the `devel/msdos/` directory of your local FTP mirror for metalab.unc.edu; Also see the MOSS DOS−extender from the Flux project from the university of Utah.

Other documents and FAQs are more DOS−centered; we do not recommend DOS development.

Windows and Co. This document is not about Windows programming, you can find lots of documents about it everywhere.. The thing you should know is that Cygnus Solutions developed the cygwin32.dll library, for GNU programs to run on Win32 platform; thus, you can use GCC, GAS, all the GNU tools, and many other Unix applications.

5.3. Your own OS

Control is what attracts many OS developers to assembly, often is what leads to or stems from assembly hacking. Note that any system that allows self−development could be qualified an "OS", though it can run "on the top" of an underlying system (much like Linux over Mach or OpenGenera over Unix).

Hence, for easier debugging purpose, you might like to develop your "OS" first as a process running on top of Linux (despite the slowness), then use the Flux OS kit (which grants use of Linux and BSD drivers in your own OS) to make it stand−alone. When your OS is stable, it is time to write your own hardware drivers if you really love that.

This HOWTO will not cover topics such as bootloader code, getting into 32−bit mode, handling Interrupts, the basics about Intel protected mode or V86/R86 braindeadness, defining your object format and calling conventions.

The main place where to find reliable information about that all, is source code of existing OSes and bootloaders. Lots of pointers are on the following webpage: http://www.tunes.org/Review/OSes.html

Chapter 6. Quick start

6.1. Introduction

Finally, if you still want to try this crazy idea and write something in assembly (if you've reached this section −− you're real assembly fan), here's what you need to start.

As you've read before, you can write for Linux in different ways; I'll show how to use *direct* kernel calls, since this is the fastest way to call kernel service; our code is not linked to any library, does not use ELF interpreter, it communicates with kernel directly.

I will show the same sample program in two assemblers, **nasm** and **gas**, thus showing Intel and AT&T syntax.

You may also want to read Introduction to UNIX assembly programming tutorial, it contains sample code for other UNIX−like OSes.

6.1.1. Tools you need

First of all you need assembler (compiler) −− **nasm** or **gas**.

Second, you need a linker −− **ld**, since assembler produces only object code. Almost all distributions have gas and ld, in the binutils package.

As for nasm, you may have to download and install binary packages for Linux and docs from the nasm site; note that several distributions (Stampede, Debian, SuSe, Mandrake) already have nasm, check first.

If you're going to dig in, you should also install include files for your OS, and if possible, kernel source.

6.2. Hello, world!

6.2.1. Program layout

Linux is 32−bit, runs in protected mode, has flat memory model, and uses the ELF format for binaries.

A program can be divided into sections: `.text` for your code (read−only), `.data` for your data (read−write), `.bss` for uninitialized data (read−write); there can actually be a few other standard sections, as well as some user−defined sections, but there's rare need to use them and they are out of our interest here. A program must have at least `.text` section.

Now we will write our first program. Here is sample code:

6.2.2. NASM (hello.asm)

```
section .data                           ;section declaration
```

```
msg     db      "Hello, world!",0xa     ;our dear string
len     equ     $ - msg                 ;length of our dear string

section .text                           ;section declaration

                        ;we must export the entry point to the ELF linker or
    global _start       ;loader. They conventionally recognize _start as their
                        ;entry point. Use ld -e foo to override the default.

_start:

;write our string to stdout

        mov     edx,len ;third argument: message length
        mov     ecx,msg ;second argument: pointer to message to write
        mov     ebx,1   ;first argument: file handle (stdout)
        mov     eax,4   ;system call number (sys_write)
        int     0x80    ;call kernel

;and exit

        mov     ebx,0   ;first syscall argument: exit code
        mov     eax,1   ;system call number (sys_exit)
        int     0x80    ;call kernel
```

6.2.3. GAS (hello.S)

```
.data                                   # section declaration

msg:
        .string "Hello, world!\n"       # our dear string
        len = . - msg                   # length of our dear string

.text                                   # section declaration

                        # we must export the entry point to the ELF linker or
    .global _start      # loader. They conventionally recognize _start as their
                        # entry point. Use ld -e foo to override the default.

_start:

# write our string to stdout

        movl    $len,%edx       # third argument: message length
        movl    $msg,%ecx       # second argument: pointer to message to write
        movl    $1,%ebx         # first argument: file handle (stdout)
        movl    $4,%eax         # system call number (sys_write)
        int     $0x80           # call kernel

# and exit

        movl    $0,%ebx         # first argument: exit code
        movl    $1,%eax         # system call number (sys_exit)
        int     $0x80           # call kernel
```

6.3. Building an executable

6.3.1. Producing object code

First step of building an executable is compiling (or assembling) object file from the source:

For nasm example:

```
$ nasm -f elf hello.asm
```

For gas example:

```
$ as -o hello.o hello.S
```

This makes `hello.o` object file.

6.3.2. Producing executable

Second step is producing executable file itself from the object file by invoking linker:

```
$ ld -s -o hello hello.o
```

This will finally build `hello` executable.

Hey, try to run it... Works? That's it. Pretty simple.

Chapter 7. Resources

7.1. Pointers

Your main resource for Linux/UNIX assembly programming material is:

> http://linuxassembly.org/resources.html

Do visit it, and get plenty of pointers to assembly projects, tools, tutorials, documentation, guides, etc, concerning different UNIX operating systems and CPUs. Because it evolves quickly, I will no longer duplicate it here.

If you are new to assembly in general, here are few starting pointers:

- The Art Of Assembly
- x86 assembly FAQ
- ftp.luth.se mirrors the hornet and x2ftp former archives of msdos assembly coding stuff
- CoreWars, a fun way to learn assembly in general
- Usenet: comp.lang.asm.x86; alt.lang.asm

7.2. Mailing list

If you're are interested in Linux/UNIX assembly programming (or have questions, or are just curious) I especially invite you to join Linux assembly programming mailing list.

This is an open discussion of assembly programming under Linux, *BSD, BeOS, or any other UNIX/POSIX like OS; also it is not limited to x86 assembly (Alpha, Sparc, PPC and other hackers are welcome too!).

Mailing list address is `<linux-assembly@vger.kernel.org>`.

To subscribe send a messgage to `<majordomo@vger.kernel.org>` with the following line in the body of the message:

```
subscribe linux-assembly
```

Detailed information and list archives are available at http://linuxassembly.org/list.html.

Chapter 8. Frequently Asked Questions

Here are frequently asked questions (with answers) about Linux assembly programming. Some of the questions (and the answers) were taken from the the linux−assembly mailing list.

8.1. How do I do graphics programming in Linux?

An answer from Paul Furber:

```
Ok you have a number of options to graphics in Linux. Which one you use
depends on what you want to do. There isn't one Web site with all the
information but here are some tips:

SVGALib: This is a C library for console SVGA access.
Pros: very easy to learn, good coding examples, not all that different
from equivalent gfx libraries for DOS, all the effects you know from DOS
can be converted with little difficulty.
Cons: programs need superuser rights to run since they write directly to
the hardware, doesn't work with all chipsets, can't run under X−Windows.
Search for svgalib−1.4.x on http://ftp.is.co.za

Framebuffer: do it yourself graphics at SVGA res
Pros: fast, linear mapped video access, ASM can be used if you want :)
Cons: has to be compiled into the kernel, chipset−specific issues, must
switch out of X to run, relies on good knowledge of linux system calls
and kernel, tough to debug
Examples: asmutils (http://www.linuxassembly.org) and the leaves example
and my own site for some framebuffer code and tips in asm
(http://ma.verick.co.za/linux4k/)

Xlib: the application and development libraries for XFree86.
Pros: Complete control over your X application
Cons: Difficult to learn, horrible to work with and requires quite a bit
of knowledge as to how X works at the low level.
Not recommended but if you're really masochistic go for it. All the
include and lib files are probably installed already so you have what
you need.

Low−level APIs: include PTC, SDL, GGI and Clanlib
Pros: very flexible, run under X or the console, generally abstract away
the video hardware a little so you can draw to a linear surface, lots of
good coding examples, can link to other APIs like OpenGL and sound libs,
Windows DirectX versions for free
Cons: Not as fast as doing it yourself, often in development so versions
can (and do) change frequently.
Examples: PTC and GGI have excellent demos, SDL is used in sdlQuake,
Myth II, Civ CTP and Clanlib has been used for games as well.

High−level APIs: OpenGL − any others?
```

```
Pros: clean api, tons of functionality and examples, industry standard
so you can learn from SGI demos for example
Cons: hardware acceleration is normally a must, some quirks between
versions and platforms
Examples: loads - check out www.mesa3d.org under the links section.

To get going try looking at the svgalib examples and also install SDL
and get it working. After that, the sky's the limit.
```

8.2. How do I debug pure assembly code under Linux?

There's an early version of the Assembly Language Debugger, which is designed to work with assembly code, and is portable enough to run on Linux and *BSD. It is already functional and should be the right choice, check it out!

You can also try **gdb** ;). Although it is source−level debugger, it can be used to debug pure assembly code, and with some trickery you can make **gdb** to do what you need (unfortunately, nasm '−g' switch does not generate proper debug info for gdb; this is nasm bug, I think). Here's an answer from Dmitry Bakhvalov:

```
Personally, I use gdb for debugging asmutils. Try this:

1) Use the following stuff to compile:
   $ nasm −f elf −g smth.asm
   $ ld −o smth smth.o

2) Fire up gdb:
   $ gdb smth

3) In gdb:
   (gdb) disassemble _start
   Place a breakpoint at _start+1 (If placed at _start the breakpoint
   wouldnt work, dunno why)
   (gdb) b *0x8048075

   To step thru the code I use the following macro:
   (gdb)define n
   >ni
   >printf "eax=%x ebx=%x ...etc...",$eax,$ebx,...etc...
   >disassemble $pc $pc+15
   >end

   Then start the program with r command and debug with n.

   Hope this helps.
```

An additional note from ???:

```
    I have such a macro in my .gdbinit for quite some time now, and it
    for sure makes life easier. A small difference : I use "x /8i $pc",
    which guarantee a fixed number of disassembled instructions. Then,
    with a well chosen size for my xterm, gdb output looks like it is
    refreshed, and not scrolling.
```

If you want to set breakpoints across your code, you can just use `int 3` instruction as breakpoint (instead of entering address manually in **gdb**).

If you're using gas, you should consult gas and gdb related tutorials.

8.3. Any other useful debugging tools?

Definitely **strace** can help a lot (**ktrace** and **kdump** on FreeBSD), it is used to trace system calls and signals. Read its manual page (**man strace**) and **strace --help** output for details.

8.4. How do I access BIOS functions from Linux (BSD, BeOS, etc)?

Noway. This is protected mode, use OS services instead. Again, you can't use `int 0x10`, `int 0x13`, etc. Fortunately almost everything can be implemented by means of system calls or library functions. In the worst case you may go through direct port access, or make a kernel patch to implement needed functionality.

8.5. Is it possible to write kernel modules in assembly?

Yes, indeed it is. While in general it is not a good idea (it hardly will speedup anything), there may be a need of such wizardy. The process of writing a module itself is not that hard -- a module must have some predefined global function, it may also need to call some external functions from the kernel. Examine kernel source code (that can be built as module) for details.

Meanwhile, here's an example of a minimum dumb kernel module (`module.asm`) (source is based on example by mammon_ from APJ #8):

```
section .text

        global init_module
        global cleanup_module
        global kernel_version

        extern printk

init_module:
        push    dword str1
        call    printk
        pop     eax
        xor     eax,eax
        ret

cleanup_module:
        push    dword str2
        call    printk
        pop     eax
        ret

str1            db      "init_module done",0xa,0
str2            db      "cleanup_module done",0xa,0

kernel_version  db      "2.2.18",0
```

The only thing this example does is reporting its actions. Modify `kernel_version` to match yours, and build module with:

```
$ nasm -f elf -o module.m module.asm
```

```
$ ld -r -o module.o module.m
```

Now you can play with it using **insmod/rmmod/lsmod** (root privilidged are required); a lot of fun, huh?

8.6. How do I allocate memory dynamically?

A laconic answer from H−Peter Recktenwald:

```
        ebx := 0        (in fact, any value below .bss seems to do)
        sys_brk
        eax := current top (of .bss section)

        ebx := [ current top < ebx < (esp - 16K) ]
        sys_brk
        eax := new top of .bss
```

An extensive answer from Tiago Gasiba:

```
section .bss

var1    resb    1

section .text

;
;allocate memory
;

%define LIMIT   0x4000000                       ; about 100Megs

        mov     ebx,0                           ; get bottom of data segment
        call    sys_brk

        cmp     eax,-1                          ; ok?
        je      erro1

        add     eax,LIMIT                       ; allocate +LIMIT memory
        mov     ebx,eax
        call    sys_brk

        cmp     eax,-1                          ; ok?
        je      erro1

        cmp     eax,var1+1                      ; has the data segment grown?
        je      erro1

;
;use allocated memory
;
                                                ; now eax contains bottom of
                                                ; data segment
        mov     ebx,eax                         ; save bottom
        mov     eax,var1                        ; eax=beginning of data segment
repeat:
        mov     word    [eax],1                 ; fill up with 1's
        inc     eax
        cmp     ebx,eax                         ; current pos = bottom?
        jne     repeat

;
;free memory
;

        mov     ebx,var1                        ; deallocate memory
```

```
        call    sys_brk                         ; by forcing its beginning=var1

        cmp     eax,-1                          ; ok?
        je      erro2
```

8.7. I can't understand how to use `select` system call!

An answer from Patrick Mochel:

```
When you call sys_open, you get back a file descriptor, which is simply an
index into a table of all the open file descriptors that your process has.
stdin, stdout, and stderr are always 0, 1, and 2, respectively, because
that is the order in which they are always open for your process from there.
Also, notice that the first file descriptor that you open yourself (w/o first
closing any of those magic three descriptors) is always 3, and they increment
from there.

Understanding the index scheme will explain what select does. When you
call select, you are saying that you are waiting certain file descriptors
to read from, certain ones to write from, and certain ones to watch from
exceptions from. Your process can have up to 1024 file descriptors open,
so an fd_set is just a bit mask describing which file descriptors are valid
for each operation. Make sense?

Since each fd that you have open is just an index, and it only needs to be
on or off for each fd_set, you need only 1024 bits for an fd_set structure.
1024 / 32 = 32 longs needed to represent the structure.

Now, for the loose example.
Suppose you want to read from a file descriptor (w/o timeout).

- Allocate the equivalent to an fd_set.

.data

my_fds: times 32 dd 0

- open the file descriptor that you want to read from.

- set that bit in the fd_set structure.

   First, you need to figure out which of the 32 dwords the bit is in.

   Then, use bts to set the bit in that dword. bts will do a modulo 32
   when setting the bit. That's why you need to first figure out which
   dword to start with.

   mov edx, 0
   mov ebx, 32
   div ebx

   lea ebx, my_fds
   bts ebx[eax * 4], edx

- repeat the last step for any file descriptors you want to read from.

- repeat the entire exercise for either of the other two fd_sets if you want action from them.

That leaves two other parts of the equation - the n paramter and the timeout
parameter. I'll leave the timeout parameter as an exercise for the reader
(yes, I'm lazy), but I'll briefly talk about the n parameter.
```

```
It is the value of the largest file descriptor you are selecting from (from
any of the fd_sets), plus one. Why plus one? Well, because it's easy to
determine a mask from that value. Suppose that there is data available on
x file descriptors, but the highest one you care about is (n - 1). Since
an fd_set is just a bitmask, the kernel needs some efficient way for
determining whether to return or not from select. So, it masks off the bits
that you care about, checks if anything is available from the bits that are
still set, and returns if there is (pause as I rummage through kernel source).
Well, it's not as easy as I fantasized it would be. To see how the kernel
determines that mask, look in fs/select.c in the kernel source tree.

Anyway, you need to know that number, and the easiest way to do it is to save
the value of the last file descriptor open somewhere so you don't lose it.

Ok, that's what I know. A warning about the code above (as always) is that
it is not tested. I think it should work, but if it doesn't let me know.
But, if it starts a global nuclear meltdown, don't call me. ;-)
```

That's all for now, folks.

Appendix A. History

Each version includes a few fixes and minor corrections, that need not to be repeatedly mentioned every time.

Revision History

Revision 0.6d	18 Mar 2001	Revised by: konst
Added Free Pascal; new NASM URL again		
Revision 0.6c	15 Feb 2001	Revised by: konst
Added SHASM; new answer in FAQ, new NASM URL, new mailing list address		
Revision 0.6b	21 Jan 2001	Revised by: konst
new questions in FAQ, corrected few URLs		
Revision 0.6a	10 Dec 2000	Revised by: konst
Remade section on AS86 (thanks to Holluby Istvan for pointing out obsolete information). Fixed several URLs that can be incorrectly rendered from sgml to html.		
Revision 0.6	11 Nov 2000	Revised by: konst
HOWTO is completely rewritten using DocBook DTD. Layout is totally rearranged; too much changes to list them here.		
Revision 0.5n	07 Nov 2000	Revised by: konst
Added question regarding kernel modules to FAQ, fixed NASM URLs, GAS has Intel syntax too		
Revision 0.5m	22 Oct 2000	Revised by: konst
Linux 2.4 system calls can have 6 args, Added ALD note to FAQ, fixed mailing list subscribe address		
Revision 0.5l	23 Aug 2000	Revised by: konst
Added TDASM, updates on NASM		
Revision 0.5k	11 Jul 2000	Revised by: konst
Few additions to FAQ		
Revision 0.5j	14 Jun 2000	Revised by: konst

Complete rearrangement of Introduction and Resources sections. FAQ added to Resources, misc cleanups and additions.

Revision 0.5i	04 May 2000	Revised by: konst

Added HLA, TALC; rearrangements in Resources, Quick Start Assemblers sections. Few new pointers.

Revision 0.5h	09 Apr 2000	Revised by: konst

finally managed to state LDP license on document, new resources added, misc fixes

Revision 0.5g	26 Mar 2000	Revised by: konst

new resources on different CPUs

Revision 0.5f	02 Mar 2000	Revised by: konst

new resources, misc corrections

Revision 0.5e	10 Feb 2000	Revised by: konst

URL updates, changes in GAS example

Revision 0.5d	01 Feb 2000	Revised by: konst

Resources (former "Pointers") section completely redone, various URL updates.

Revision 0.5c	05 Dec 1999	Revised by: konst

New pointers, updates and some rearrangements. Rewrite of sgml source.

Revision 0.5b	19 Sep 1999	Revised by: konst

Discussion about libc or not libc continues. New web pointers and and overall updates.

Revision 0.5a	01 Aug 1999	Revised by: konst

Quick Start section rearranged, added GAS example. Several new web pointers.

Revision 0.5	01 Aug 1999	Revised by: konstfare

GAS has 16−bit mode. New maintainer (at last): Konstantin Boldyshev. Discussion about libc or not libc. Added Quick Start section with examples of assembly code.

Revision 0.4q	22 Jun 1999	Revised by: fare

process argument passing (argc, argv, environ) in assembly. This is yet another "last release by Fare before new maintainer takes over". Nobody knows who might be the new maintainer.

Revision 0.4p	06 Jun 1999	Revised by: fare

clean up and updates

Revision 0.4o	01 Dec 1998	Revised by: fare
Revision 0.4m	23 Mar 1998	Revised by: fare

corrections about gcc invocation

Revision 0.4l	16 Nov 1997	Revised by: fare

release for LSL 6th edition

Revision 0.4k	19 Oct 1997	Revised by: fare
Revision 0.4j	07 Sep 1997	Revised by: fare
Revision 0.4i	17 Jul 1997	Revised by: fare

info on 16−bit mode access from Linux

Revision 0.4h	19 Jun 1997	Revised by: fare

still more on "how not to use assembly"; updates on NASM, GAS.

Revision 0.4g	30 Mar 1997	Revised by: fare
Revision 0.4f	20 Mar 1997	Revised by: fare

Revision 0.4e	13 Mar 1997	Revised by: fare
Release for DrLinux		
Revision 0.4d	28 Feb 1997	Revised by: fare
Vapor announce of a new Assembly−HOWTO maintainer		
Revision 0.4c	09 Feb 1997	Revised by: fare
Added section Do you need assembly?.		
Revision 0.4b	03 Feb 1997	Revised by: fare
NASM moved: now is before AS86		
Revision 0.4a	20 Jan 1997	Revised by: fare
CREDITS section added		
Revision 0.4	20 Jan 1997	Revised by: fare
first release of the HOWTO as such		
Revision 0.4pre1	13 Jan 1997	Revised by: fare
text mini−HOWTO transformed into a full linuxdoc−sgml HOWTO, to see what the SGML tools are like		
Revision 0.3l	11 Jan 1997	Revised by: fare
Revision 0.3k	19 Dec 1996	Revised by: fare
What? I had forgotten to point to terse???		
Revision 0.3j	24 Nov 1996	Revised by: fare
point to French translated version		
Revision 0.3i	16 Nov 1996	Revised by: fare
NASM is getting pretty slick		
Revision 0.3h	06 Nov 1996	Revised by: fare
more about cross−compiling −− See on sunsite: devel/msdos/		
Revision 0.3g	02 Nov 1996	Revised by: fare
Created the History. Added pointers in cross−compiling section. Added section about I/O programming under Linux (particularly video).		
Revision 0.3f	17 Oct 1996	Revised by: fare
Revision 0.3c	15 Jun 1996	Revised by: fare
Revision 0.2	04 May 1996	Revised by: fare
Revision 0.1	23 Apr 1996	Revised by: fare
Francois−Rene "Fare" Rideau creates and publishes the first mini−HOWTO, because "I'm sick of answering ever the same questions on comp.lang.asm.x86"		

Appendix B. Acknowledgements

I would like to thank all the people who have contributed ideas, answers, remarks, and moral support, and additionally the following persons, by order of appearance:

- Linus Torvalds for Linux
- Bruce Evans for bcc from which as86 is extracted
- Simon Tatham and Julian Hall for NASM

- Greg Hankins and now Tim Bynum for maintaining HOWTOs
- Raymond Moon for his FAQ
- Eric Dumas for his translation of the mini−HOWTO into French (sad thing for the original author to be French and write in English)
- Paul Anderson and Rahim Azizarab for helping me, if not for taking over the HOWTO
- Marc Lehman for his insight on GCC invocation
- Abhijit Menon−Sen for helping me figure out the argument passing convention

---

Appendix C. Endorsements

This version of the document is endorsed by Konstantin Boldyshev.

Modifications (including translations) must remove this appendix according to the license agreement.

```
$Id: Assembly−HOWTO.sgml,v 1.24 2001/03/18 07:35:35 konst Exp $
```

Appendix D. GNU Free Documentation License

holder saying it can be distributed under the terms of this License. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you".

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front−matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (For example, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License.

The "Cover Texts" are certain short passages of text that are listed, as Front−Cover Texts or Back−Cover Texts, in the notice that says that the Document is released under this License.

A "Transparent" copy of the Document means a machine−readable copy, represented in a format whose specification is available to the general public, whose contents can be viewed and edited directly and straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup has been designed to thwart or discourage subsequent modification by readers is not Transparent. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard−conforming simple HTML designed for human modification. Opaque formats include PostScript, PDF, proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine−generated HTML produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies of the Document numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front−Cover Texts on the front cover, and Back−Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine−readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a publicly−accessible computer−network location containing a complete Transparent copy of the Document, free of added material, which the general network−using public has access to download anonymously at no charge using public−standard network protocols. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has less than five).
C. State on the Title page the name of the publisher of the Modified Version, as the publisher.
D. Preserve all the copyright notices of the Document.
E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
H. Include an unaltered copy of this License.
I. Preserve the section entitled "History", and its title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
K. In any section entitled "Acknowledgements" or "Dedications", preserve the section's title, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
M. Delete any section entitled "Endorsements". Such a section may not be included in the Modified Version.
N. Do not retitle any existing section as "Endorsements" or to conflict in title with any Invariant Section.

If the Modified Version includes new front−matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties−−for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front−Cover Text, and a passage of up to 25 words as a Back−Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front−Cover Text and one of Back−Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections entitled "History" in the various original documents, forming one section entitled "History"; likewise combine any sections entitled "Acknowledgements", and any sections entitled "Dedications". You must delete all sections entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, does not as a whole count as a Modified Version of the Document, provided no compilation copyright is claimed for the compilation. Such a compilation is called an "aggregate", and this License does not apply to the other self–contained works thus compiled with the Document, on account of their being thus compiled, if they are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one quarter of the entire aggregate, the Document's Cover Texts may be placed on covers that surround only the Document within the aggregate. Otherwise they must appear on covers around the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License provided that you also include the original English version of this License. In case of a disagreement between the translation and the original English version of this License, the original English version will prevail.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> Copyright (c) YEAR YOUR NAME.
> Permission is granted to copy, distribute and/or modify this document
> under the terms of the GNU Free Documentation License, Version 1.1
> or any later version published by the Free Software Foundation;
> with the Invariant Sections being LIST THEIR TITLES, with the
> Front−Cover Texts being LIST, and with the Back−Cover Texts being LIST.
> A copy of the license is included in the section entitled "GNU
> Free Documentation License".

If you have no Invariant Sections, write "with no Invariant Sections" instead of saying which ones are invariant. If you have no Front−Cover Texts, write "no Front−Cover Texts" instead of "Front−Cover Texts being LIST"; likewise for Back−Cover Texts.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.