Linux Assembly HOWTO

Table of Contents

Table of Contents

Linux Assembly HOWTO

Konstantin Boldyshev and Francois−Rene Rideau

v0.5l, August 23, 2000

This is the Linux Assembly HOWTO. 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. Contributions about them are gladly accepted. ***Keywords****:* `assembly, assembler, asm, inline asm, macroprocessor, preprocessor, 32−bit, IA−32, i386, x86, nasm, gas, as86, OS, kernel, system, libc, system call, interrupt, small, fast, embedded, hardware, port`

1.INTRODUCTION

You can skip this section if you are familiar with HOWTOs, or just hate to read all this assembly−nonrelated crap.

1.1 Legal Blurb

All modified documents, including translations, anthologies, and partial documents, must meet the following requirements:

- The modified version must be labeled as such
- The person making the modifications must be identified
- Acknowledgement of the original author must be retained
- The location of the original unmodified document be identified
- The original author's (or authors') name(s) may not be used to assert or imply endorsement of the resulting document without the original author's (or authors') permission

The most recent official version of this document is available from Linux Assembly and LDP sites. If you are reading a few−months−old copy, consider checking urls above 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, 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 FSF (free software foundation) 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 Assembly−HOWTO maintainer. At the time of this writing, it is Konstantin Boldyshev and no more Francois−Rene Rideau. 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.

1.4 Credits

I would like to thank 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
- All the people who have contributed ideas, answers, remarks, and moral support.

1.5 History

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

Version 0.5l 23 Aug 2000

Added TDASM, updates on NASM

Version 0.5k 11 Jul 2000

Few additions to FAQ

Version 0.5j 14 Jun 2000

Complete rearrangement of INTRODUCTION and RESOURCES; FAQ added to RESOURCES, misc cleanups and additions (and more to come)

Version 0.5i 04 May 2000

Added HLA, TALC; rearrangements in RESOURCES, QUICK START, ASSEMBLERS; few new pointers

Version 0.5h 09 Apr 2000

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

Version 0.5g 26 Mar 2000

new resources on different CPUs

Version 0.5f 02 Mar 2000

new resources, misc corrections

Version 0.5e 10 Feb 2000

url updates, changes in GAS example

Version 0.5d 01 Feb 2000

RESOURCES (former POINTERS) section completely redone, various url updates.

Version 0.5c 05 Dec 1999

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

Version 0.5b 19 Sep 1999

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

Version 0.5a 01 Aug 1999

"QUICK START" section rearranged, added GAS example. Several new web pointers.

Version 0.5 25 July 1999

GAS has 16−bit mode. New maintainer (at last): Konstantin Boldyshev. Discussion about libc or not libc. Added section "QUICK START" with examples of using assembly.

Version 0.4q 22 June 1999

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.

Version 0.4p 6 June 1999

clean up and updates.

Version 0.4o 1 December 1998

*

Version 0.4m 23 March 1998

corrections about gcc invocation

Version 0.4l 16 November 1997

release for LSL 6th edition.

Version 0.4k 19 October 1997

*

Version 0.4j 7 September 1997

*

Version 0.4i 17 July 1997

info on 16−bit mode access from Linux.

Version 0.4h 19 Jun 1997

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

Version 0.4g 30 Mar 1997

*

Version 0.4f 20 Mar 1997

*

Version 0.4e 13 Mar 1997

Release for DrLinux

Version 0.4d 28 Feb 1997

Vapor announce of a new Assembly−HOWTO maintainer.

Version 0.4c 9 Feb 1997

Added section "DO YOU NEED ASSEMBLY?"

Version 0.4b 3 Feb 1997

NASM moved: now is before AS86

Version 0.4a 20 Jan 1997

CREDITS section added

Version 0.4 20 Jan 1997

first release of the HOWTO as such.

Version 0.4pre1 13 Jan 1997

text mini−HOWTO transformed into a full linuxdoc−sgml HOWTO, to see what the SGML tools are like.

Version 0.3l 11 Jan 1997

*

Version 0.3k 19 Dec 1996

What? I had forgotten to point to terse???

Version 0.3j 24 Nov 1996

point to French translated version

Version 0.3i 16 Nov 1996

NASM is getting pretty slick

Version 0.3h 6 Nov 1996

more about cross−compiling −− See on sunsite: devel/msdos/

Version 0.3g 2 Nov 1996

Created the History. Added pointers in cross−compiling section. Added section about I/O programming under Linux (particularly video).

Version 0.3f 17 Oct 1996

*

Version 0.3c 15 Jun 1996

*

Version 0.2 04 May 1996

*

Version 0.1 23 Apr 1996

Francois−Rene "Fare" Rideau <fare@tunes.org> creates and publishes the first mini−HOWTO, because "I'm sick of answering ever the same questions on comp.lang.asm.x86"

2.DO YOU NEED ASSEMBLY?

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

2.1 Pros and Cons

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 behavior of code 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 such 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's something few will ever do, and even they, not often).

The disadvantages of Assembly

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

- it's long and tedious to write initially,
- it's quite bug−prone,
- your bugs can be very difficult to chase,
- it's very difficult to understand and modify, i.e. to maintain.
- the result is very non−portable to other architectures, existing or future,
- 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, throughput, 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, Pentium II, Pentium III; Cyrix 5x86, 6x86; AMD K5, K6 (K6−2, K6−III), K7 (Athlon). New designs keep popping up, so don't expect either this listing or 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.

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

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. "

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.

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. 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.

Inspecting compiler−generated code

There are many reasons to inspect compiler−generated assembly code. Here are 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 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

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. Shortly, main practical reasons why you may need to get into Linux assembly are *small code* and *libc independence*. Non−practical (and most often) reason is being just an old crazy hacker, who has twenty years old habit of doing everything in assembly language.

Also, if you're porting Linux to some embedded hardware you can be quite short at 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 ways you've got 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.

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.

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 precompiled 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 precompiled binaries should be found at your usual FTP sites.

For most popular DOS port of GCC is named DJGPP, and can be found in directories of such name in FTP sites. See http://www.delorie.com/djgpp/.

There are two Win32 GCC ports: cygwin and mingw

There is also a port of GCC to OS/2 named EMX, that also works under DOS, and includes lots of unix−emulation library routines. See around the following site: ftp://ftp−os2.cdrom.com/pub/os2/emx09c/.

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 follow its syntax (see below); do mind that inline asm needs

percent characters to be quoted so they 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.

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 longer, and yields code that is a lot larger, but only a bit faster; such overoptimization 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 warnings and helps you 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 computer. 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 *your* system can be found by asking `gcc −v`.

3.2 GAS

GAS is the GNU Assembler, that GCC relies upon.

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/.

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 no worse, no 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 ax,dx` (move contents of register `dx` into register `ax`) will be in GAS syntax `mov %dx, %ax`.
- The operand length 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 `movw %dx,%ax`. However, gas does not require strict AT&T syntax, so the suffix is optional when length 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`).
- No prefix to an operand indicates it is a memory−address; hence `movl $foo,%eax` puts the *address* of variable `foo` in register `%eax`, but `movl foo,%eax` puts the *contents* of variable

`foo` in 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`).

A program exists to help you convert programs from TASM syntax to AT&T syntax. See ftp://x2ftp.oulu.fi/pub/msdos/programming/convert/ta2asv08.zip. (Since the original x2ftp site is closing (no more?), use a mirror site). There also exists a program for the reverse conversion: http://www.multimania.com/placr/a2i.html.

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, 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.

16−bit mode

The current stable release of binutils (2.9.1.0.25) now fully supports 16−bit mode (registers *and* addressing) on i386 PCs. Still with its peculiar AT&T syntax, of course. Use `.code16` and `.code32` to switch between assembly modes.

Also, a neat trick used by some (including the oskit authors) is to have GCC 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.

GASP

GASP is the GAS Preprocessor. It adds macros and some nice syntax to GAS. GASP comes together with GAS in the GNU binutils archive. It works as a filter, much like cpp and the like. I have no idea on details, but it comes with its own texinfo documentation, so just browse them (in .info), print them, grok them. 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.

Where to find NASM

http://nasm.sourceforge.net

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

What it does

The syntax is Intel−style. Fairly good macroprocessing support is integrated.

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

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 is actively supported online, and runs on all platforms.

Note: NASM also 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 converters between GAS AT&T and Intel assembler syntax, which perform conversion in both directions.

3.4 AS86

AS86 is a 80x86 assembler, both 16−bit and 32−bit, part of Bruce Evans' C Compiler (BCC). It has mostly Intel−syntax, though it differs slightly as for addressing modes.

Where to get AS86

A completely outdated version of AS86 is distributed by HJLu just to compile the Linux kernel, in a package named bin86 (current version 0.4), 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.

The most recent versions by Bruce Evans (bde@zeta.org.au) are published together with the FreeBSD distribution. Well, they were: I could not find the sources from distribution 2.1 on :(Hence, I put the sources at my place: http://www.tunes.org/~fare/files/asm/bcc−95.3.12.src.tgz

The Linux/8086 (aka ELKS) project is somehow maintaining bcc (though I don't think they included the 32−bit patches). See around http://www.linux.org.uk/ELKS−Home/ (or http://www.elks.ecs.soton.ac.uk) and ftp://linux.mit.edu/pub/linux/ELKS/. I haven't followed these developments, and would appreciate a reader contributing on this topic.

Among other things, these more recent versions, unlike HJLu's, supports Linux GNU a.out format, so you can link you code to Linux programs, and/or use the usual tools from the GNU binutils package to manipulate your data. This version can co−exist without any harm with the previous one (see according question below).

BCC from 12 march 1995 and earlier version has a misfeature that makes all segment pushing/popping 16−bit, which is quite annoying when programming in 32−bit mode. I wrote a patch at a time when the TUNES Project used as86: http://www.tunes.org/~fare/files/asm/as86.bcc.patch.gz. Bruce Evans accepted this patch, but since as far as I know he hasn't published a new release of bcc, the ones to ask about integrating it (if not done yet) are the ELKS developers.

How to invoke the assembler?

Here's the GNU Makefile entry for using bcc to transform `.s` asm into both GNU 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 GNU a.out, you can see the docs of bcc about the other supported formats, and/or use the objcopy utility from the GNU binutils package.

Where to find docs

The docs are what is included in the bcc package. I salvaged the man pages that used to be available from the FreeBSD site at http://www.tunes.org/~fare/files/asm/bcc−95.3.12.src.tgz. Maybe ELKS developers know

better. When in doubt, the sources themselves are often a good docs: it's not very well commented, but the programming style is straightforward. You might try to see how as86 is used in ELKS or Tunes 0.0.0.25...

What if I can't compile Linux anymore with this new version ?

Linus is buried alive in mail, and since HJLu (official bin86 maintainer) chose to write hacks around an obsolete version of as86 instead of building clean code around the latest version, I don't think my patch for compiling Linux with a modern as86 has any chance to be accepted if resubmitted. Now, this shouldn't matter: just keep your as86 from the bin86 package in `/usr/bin/`, and let bcc install the good as86 as `/usr/local/libexec/i386/bcc/as` where it should be. You never need explicitly call this "good" as86, because bcc does everything right, including conversion to Linux a.out, when invoked with the right options; so assemble files exclusively with bcc as a frontend, not directly with as86.

Since GAS now supports 16−bit code, and since H. Peter Anvin, well−known linux hacker, works on NASM, maybe Linux will get rid of AS86, anyway? Who knows!

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).

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/.

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/.

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"

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.

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.

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

You may find more about them, together with the basics of x86 assembly programming, in 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.

4.METAPROGRAMMING/MACROPROCESSING

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's not fit; C, OCaml, perl, Scheme, might be a better choice for most of your programming.

However, there are cases when these tools do not give a fine enough control on the machine, and assembly is useful or needed. In those case, you'll appreciate a system of macroprocessing and metaprogramming that'll allow recurring patterns to be factored each into a 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 What's integrated into the above

Yes I know this section does not contain much useful up−to−date information. Feel free to contribute what you discover the hard way...

GCC

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.

GAS

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 to feed them to GAS. Again and again, see Linux sources for examples.

GASP

It adds all the usual macroassembly tricks to GAS. See its texinfo docs.

NASM

NASM has comprehensive macro support, too. See according docs. If you have some bright idea, you might wanna contact the authors, as they are actively developing it. Meanwhile, see about external filters below.

AS86

It has some simple macro support, but I couldn't find docs. Now the sources are very straightforward, so if you're interested, you should understand them easily. If you need more than the basics, you should use an external filter (see below).

OTHER ASSEMBLERS

- Win32FORTH: CODE and END−CODE are normal that do not switch from interpretation mode to compilation mode, so you have access to the full power of FORTH while assembling.
- TUNES: it doesn't work yet, but the Scheme language is a real high−level language that allows arbitrary meta−programming.

4.2 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) < $< > $@
```

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,

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're not writing million−lines assembly programs, are you?).

Macroprocessing with your own filter

You can write your own simple macro−expansion filter with the usual tools: perl, awk, sed, etc. That's quick to do, and you control everything. But of course, any power in macroprocessing must be earned the hard way.

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!

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!

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/

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.

5.CALLING CONVENTIONS

5.1 Linux

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 it's result 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 in the GCC info pages the section: `C Extensions::Extended Asm::`. See also how Linux defines its asmlinkage macro...

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 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 utility `objcopy`, 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.

Direct Linux syscalls

Often you will be told that using libc is the only way, and direct system calls are bad. This is true. To some extent. So, you must know that libc is not sacred, and in *most* cases libc 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 also result in improved performance, 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, modern libc becomes slower and slower, 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:

- smallest possible size; squeezing the last byte out of the system.
- 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 a 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 waste memory, not save 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.
- Consequently, 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 your benefitting 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 are 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 (as documented in section 2 of the manual pages), 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 five) in `ebx, ecx, edx, esi, edi` respectively. Result is returned in `eax`, with a negative result being an error, whose opposite is what libc would put in `errno`. The user−stack is not touched, so you needn't have a valid one when doing a syscall.

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/`).

Hardware I/O under Linux

If you want to do direct I/O under Linux, either it's something very simple that needn't 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.

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

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 university of Utah.

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

5.3 Windows and Co.

This HOWTO 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.

5.4 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 standalone. 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 Boot loader 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

6.QUICK START

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), I'll herein provide what you will need to get started.

As you've read before, you can write for Linux in different ways; I'll show example of using pure system calls. This means that we will not use libc at all, the only thing required for our program to run is kernel. Our code will not be linked to any library, will not use ELF interpreter −− it will communicate directly with kernel.

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 Tools you need

First of all you need assembler (compiler): `nasm` or `gas`. Second, you need linker: `ld`, assembler produces only object code. Almost all distributions include `gas` and `ld`, in binutils package. As for `nasm`, you may have to download and install binary packages for Linux and docs from the nasm webpage; however, several distributions (Stampede, Debian, SuSe) already include it, check first.

If you are going to dig in, you should also install kernel source. I assume that you are using at least Linux 2.0 and ELF.

6.2 Hello, world!

Linux is 32bit and has flat memory model. A program can be divided into sections. Main sections are *.text* for your code, *.data* for your data, *.bss* for undefined data. Program must have at least *.text* section.

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

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
```

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 Producing object code

First step of building binary is producing object file from source by invoking assembler; we must issue the following:

For `nasm` example:

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

For `gas` example:

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

This will produce `hello.o` object file.

6.4 Producing executable

Second step is producing executable file itself from 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.

7.RESOURCES

You main resource for Linux/UNIX assembly programming material is **Linux Assembly resources page**. 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 in this HOWTO.

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

7.1 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, FreeBSD, 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!).

List address is mailto:linux−assembly@egroups.com.

To subscribe send a blank message to mailto:linux−assembly@egroups.com.

List archives are available at http://www.egroups.com/list/linux−assembly/.

7.2 Frequently asked questions (with answers)

Here are frequently asked questions. Answers are taken from the linux−assembly mailing list.

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.
```

How do I debug pure assembly code under Linux?

If you're using `gas`, you should consult Linux assembly Tutorial by Bjorn Chambless.

With `nasm` situation is a bit different, since it doesnot support gdb specific debugging extensions. Although `gdb` 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. 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`).

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.

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 through 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.

That's all for now, folks.

$Id: Assembly−HOWTO.sgml,v 1.17 2000/08/23 12:41:06 konst Exp $
