Assembly HOWTO

François-René Rideau rideau@ens.fr v0.4g, 30 March 1997

This is the Linux Assembly HOWTO. This document describes how to program in assembly using *FREE* programming tools, focusing on development for or from the Linux Operating System an i386 platforms. Included material may or may not be applicable to other hardware and/or software platforms. Contributions about these would be gladly accepted. *keywords*: assembly, assembler, free, macroprocessor, preprocessor, asm, inline asm, 32-bit, x86, i386, gas, as86, nasm

Contents

1 INTRODUCTION

1.1 Legal Blurp

1.2 IMPORTANT NOTE

This is expectedly the last release I'll make of this document. There's one candidate new maintainer, but until he really takes the HOWTO over, I'll accept feedback.

You are especially invited to ask questions, to answer to questions, to correct given answers, to add new FAQ answers, to give pointers to other software, to point the current maintainer to bugs or deficiencies in the pages. If you're motivated, you could even *TAKE OVER THE MAINTENANCE OF THE FAQ*. In one word, contribute!

To contribute, please contact whoever appears to maintain the Assembly-HOWTO, currently me (`rideau@clipper.ens.fr`) but hopefully soon Rahim Azizarab (`rahim@megsinet.net`).

1.3 Foreword

This document aims at answering frequently asked questions of people who program or want to program 32-bit x86 assembly using *free* assemblers, particularly under the Linux operating system. It may also point to other documents about non-free, non-x86, or non-32-bit assemblers, though it is not its primary goal.

Because the main interest of assembly programming is to build to write the guts of operating systems, languages, and games, where a C compiler fails to provide the needed expressivity (performance is more and more seldom an issue), we stress on development of such software.

1.3.1 How to use this document

This document contains answers to some frequently asked questions. At many places, Universal Resource Locators (URL) are given for some software or documentation repository. Please see that the most useful repositories are mirrored, and that by accessing a nearer mirror site, you relieve the whole Internet from unneeded network traffic, while saving your own precious time. Particularly, there are large repositories all over the world, that mirror other popular repositories. You should learn and note what are those places near you (networkwise). Sometimes, the list of mirrors is listed in a file, or in a login message. Please heed the advice. Else, you should ask archie about the software you're looking for...

The most recent version for this documents sits in

http://www.eleves.ens.fr:8080/home/rideau/Assembly-HOWTO or
http://www.eleves.ens.fr:8080/home/rideau/Assembly-HOWTO.sgml

but what's in Linux HOWTO repositories *should* be fairly up to date, too (I can't know):

ftp://sunsite.unc.edu/pub/Linux/docs/HOWTO/ (?)

A french translation of this HOWTO or an earlier version of it might sit around

ftp://ftp.ibp.fr/pub/linux/french/HOWTO/

1.3.2 Other related documents

- 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 a model for most; it generally comes in a file named `COPYING`, with a library version in a file named `COPYING.LIB`. Litterature from the FSF (free software foundation) might help you, too.
- Particularly, the interesting kind of free software comes with sources that you can consult and correct, or sometimes even borrow from. Read your particular license carefully, and do comply to it.
- There is a FAQ for comp.lang.asm.x86 that answers generic questions about x86 assembly programming, and questions about some commercial assemblers in a 16-bit DOS environment. Some of it apply to free 32-bit asm programming, so you may want to read this FAQ...

  http://www2.dgsys.com/~raymoon/faq/asmfaq.zip
- FAQs and docs exist about programming on your favorite platform, whichever it is, that you should consult for platform-specific issues not directly related to programming in assembler.

1.4 History

Each version includes a few fixes and minor corrections, which needs not be mentionned each time.

Version 0.1 ? ? 1996

```
Francois-Rene "Far\'{e}" Rideau {$<$}rideau@ens.fr{$>$}
creates and publishes the first mini-HOWTO,
'cause ``I'm sick of answering ever the same questions
 on comp.lang.asm.x86''
```

Version 0.2 ? ? 1996

```
?
```

Version 0.3 ? ? 1996

```
?
```

Version 0.3c 15 Jun 1996

```
?
```

Version 0.3f 17 Oct 1996

```
* found -fasm option to enable GCC inline assembler
      w/o -O optimizations
```

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 0.3h 6 Nov 1996

```
* more about cross-compiling -- See on sunsite: devel/msdos/
```

Version 0.3i 16 Nov 1996

```
* NASM is getting pretty slick
```

Version 0.3j 24 Nov 1996

```
* point to french version
```

Version 0.3k 19 Dec 1996

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

Version 0.3l 11 Jan 1997

```
?
```

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 0.4 20 Jan 1997

```
first release of the HOWTO as such.
```

Version 0.4a 20 Jan 1997

```
* CREDITS section added
```

Version 0.4b 3 Feb 1997

```
* NASM put before AS86
```

Version 0.4c 9 Feb 1997

```
* Added section "DO YOU NEED ASSEMBLY?"
```

Version 0.4d 28 Feb 1997

```
* Announcing Rahim Azizarab <rahim@megsinet.net>
as new Assembly-HOWTO maintainer
```

Version 0.4e 13 Mar 1997

```
?
```

Version 0.4f 20 Mar 1997

```
?
```

Version 0.4g 30 Mar 1997

```
* Final release by Far\'{e} (?)
```

1.5 Credits

I would like to thanks the following persons, by order of appearance:

- *Linus Torvalds* `<mailto:buried.alive@in.mail>` for Linux
- *Bruce Evans* `<mailto:bde@zeta.org.au>` for bcc from which as86 is extracted
- *Janes Faber* `<mailto:J.A.Faber@fys.ruu.nl>` for his *WWW page* `<http://www.fys.ruu.nl/~faber/Amain.html>`
- *Simon Tatham* `<mailto:sgt20@blue.csi.cam.ac.uk>`, *Julian Hall* `<mailto:csusb@csv.warwick.ac.uk>` and the other NASM hackers (for NASM, what else?)
- *Jim Neil* `<mailto:jim-neil@digital.net>` for Terse
- *Greg Hankins* `<mailto:gregh@sunsite.unc.edu>` for maintaining HOWTOs
- *Raymond Moon* `<mailto:raymoon@moonware.dgsys.com>` for his FAQ
- *Michael Taeschner* `<mailto:Michael.Taeschner@dlr.de>` for pointing me to EMX
- *KiSung Um* `<mailto:nowlinux@soback.kornet.nm.kr>` for his moral support
- *Eric Dumas* `<mailto:dumas@excalibur.ibp.fr>` for his translation of the mini-HOWTO into french...
- People I've forgot to mention – please remind me!

2 DO YOU NEED ASSEMBLY?

Well, I wouldn't want to interfere with what you're doing, but here are a few advice from 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 behavior of code in critical sections that might involve hardware or I/O lock-ups
- you can break the conventions of your usual compiler, which might allow some optimizations (like temporarily breaking rules about GC, threading, etc).

- get access to unusual programming modes of your processor (e.g. 16 bit code for startup or BIOS interface on Intel PCs)
- 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 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 that's 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).

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's long and tedious to write initially,
- it's very bug-prone,
- your bugs will 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 variation (bus width, cache speed and size, presence of FPU, MMX or other extensions) implies potentially completely different optimization techniques. CPU designs already include Intel 386, 486, Pentium, PPro; Cyrix 5x86, 6x86; AMD K5; and new designs keep appearing.
- your code might also be unportable accross different OS platforms on the same architecture, by lack of proper tools (well, NASM seems to work or be workable on all intel platforms).
- 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.
  $e.g.youmightbuildveryfastlistmanipulationprimitivesinassembly; onlyahashtablewouldhavespedupyourprogrammuchmore; or, inanothercontext, abinarytree; orsomestructuredistributedoveraclusterofCPUs$
- 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

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 (from "macros" to a high-level language)
- have automatic tools translate these programs into assembly code
- have this code be optimized if possible
- All of the above, i.e. write 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 sources for Linux (the OS) about it: as few assembly as needed, resulting in a fast, reliable, portable, maintainable OS. Even a successful game like DOOM was 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 Languages with optimizing compilers

For instance, languages like ObjectiveCAML, SML, CommonLISP, Scheme, ADA, Pascal, C, C++, all have free optimizing compilers that'll 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 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 are suches. Speed is fairly good.

2.2.2 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 instead of interpreting it;
- then try to enable 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 pipelined architectures! Slow parts of a program might be intrinsically so. Perhaps a completely different approach to the problem might help, then.

2.2.3 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
- 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!

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

ftp://prep.ai.mit.edu/pub/gnu/

together with all the released application software from the GNU project. However, there exists a lot of mirrors.

However, sources adapted to your favorite OS, and binaries precompiled for it, should be found at your usual FTP sites.

For GCC under Linux, see around

http://www.linux.org.uk/

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 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:

http://www.leo.org/pub/comp/os/os2/gnu/emx+gcc/

http://warp.eecs.berkeley.edu/os2/software/shareware/emx.html

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, that you can convert to tex, compile (with tex), and print, convert to interactive emacs .info format and browse, 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`, `%ebp` respectively (no letter for `%esp`).

A URL for this document and section, as converted in HTML format, is

http://www.cygnus.com/doc/usegcc_89.html#SEC92

The DJGPP Games resource (not only for game hackers) has this page specifically about assembly:

http://www.rt66.com/~brennan/djgpp/djgpp_asm.html

Finally, there is a web page called, "DJGPP Quick ASM Programming Guide", that covers URLs to FAQs, AT&T x86 ASM Syntax, Some inline ASM information, and converting .obj/.lib files:

http://remus.rutgers.edu/~avly/djasm.html

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 free Linux OS.

3.1.3 Invoking GCC to have it properly inline assembly code ?

Be sure to invoke GCC with the `-O` flag (or `-O2`, `-O3`, etc), to enable optimizations and inline assembly. If you don't, your code may compile, but not run properly!!! Actually (kudos to Tim Potter, timbo@moshpit.air.net.au), it is enough to use the `-fasm` flag, which is part of all the features enabled by ¡/-O/. So if you have problems with buggy optimizations in your particular implementation/version of GCC, you can still use inline asm. Similarly, use -fno-asm to disable inline assembly (why would you?).

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

```
        gcc -O2 -fomit-frame-pointer -m386
```

`-O2` is the good optimization level. Optimizing besides it yields code that is a lot larger, but only a bit faster; such overoptimizationn might be useful for tight loops only (if any), which you may be doing in assembly anyway; if you need that, do it just for the few routines that need it.

`-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.

`-m386` yields more compact code, without any measurable slowdown, (note that small code also means less disk I/O and faster execution) but perhaps on the above-mentioned tight loops; you might appreciate -mpentium for special pentium-optimizing GCC targetting a specifically pentium platform.

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...

Note that you can add make these flags the default by editing file `/usr/lib/gcc-lib/i486-linux/2.7.2.1/specs` or wherever that is on your system.

3.2 GAS

GAS is the GNU Assembler, that GCC relies upon, with

3.2.1 Where to find it

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

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

A file gas.doc or as.doc (still around the same place as you found GAS, if not in the GAS source package itself) describes the syntax. Of course, the ultimate documentation is the sources themselves!

One place for it is in FTP directory

ftp://sunsite.unc.edu/pub/Linux/GCC/

ftp://sunsite.doc.ic.ac.uk/packages/linux/sunsite.unc-mirror/GCC/ (?)

Again, the sources for Linux (the OS kernel), come in as good examples; see under linux/arch/i386, the following files: `kernel/entry.S, kernel/head.S, boot/compressed/head.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 (QT, 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 2 above.

3.2.3 Limited 16-bit mode

GAS is a 32-bit assembler, meant to support a 32-bit compiler. It currently has only limited support for 16-bit mode, which consists in prepending the 32-bit prefixes to instructions, so you write 32-bit code that runs in 16-bit mode on a 32 bit CPU. In both modes, it supports 16-bit register usage, but what is unsupported is 16-bit addressing. Use the directive `code16` and `code32` to switch between modes. Note that an inline assembly statement `asm("code16\n")` will allow GCC to produce 32-bit code that'll run in real mode! Feel free to add full 16-bit support if you think you need it, by modifying GAS. A cheaper solution is to define macros (see below) for just the 16-bit mode instructions you need (almost nothing if you use code16 as above, and can safely assume the code will run on a 32-bit capable x86 CPU). To find the proper encoding, you can get inspiration from the sources of 16-bit capable assemblers for the encoding.

3.3 GASP

GASP is the GAS Preprocessor. It adds macros and some nice syntax to GAS.

3.3.1 Where to find GASP

GASP comes together with GAS in the GNU binutils archive.

3.3.2 How it works

I have no idea, but it comes with its own texinfo documentation, so just print them, or browse the .info files... Looks like a regular macro-assembler to me.

3.4 NASM

The Netwide Assembler project is producing yet another assembler, written in C, that should be modular enough to eventually support all known syntaxes and object formats.

3.4.1 Where to find NASM

http://www.dcs.warwick.ac.uk/~jules/nasm1.html

Binary release on your usual sunsite mirror in devel/lang/asm/

3.4.2 What it does

At the time this HOWTO is written, current NASM version is 0.93.

The syntax is very simple (simplified MASM style). Little integrated macroprocessing.

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).

Surely NASM evolves too fast for this HOWTO to be kept up to date. Soon, perhaps even now, you should use NASM instead of AS86, because NASM is supported and AS86 not so much, unless you also use BCC as a 16-bit compiler package, which is out of scope of this 32-bit HOWTO. NASM online support is reportedly quite good.

Note: NASM also comes with a disassembler, NDISASM.

3.5 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.

3.5.1 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 0.3), 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, and has a few bugs in 32-bit mode, so you better keep it only for compiling Linux.

The most recent versions 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 in

http://www.eleves.ens.fr:8080/home/rideau/files/bcc-95.3.12.src.tgz

Among other things, it supports Linux GNU a.out format, so you can link you code to Linux programs, and/or use the usual tools from the GNU binutil package to manipulate your data. This version can co-exist without any harm with the previous one (see question 2.4.4 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. A patch is published in the Tunes project

http://www.eleves.ens.fr:8080/home/rideau/Tunes/

subpage `files/tgz/tunes.0.0.0.25.src.tgz` in unpacked subdirectory `LLL/i386/` The patch should also be in available directly from http://www.eleves.ens.fr:8080/home/rideau/files/as86.bcc.patch.gz Bruce Evans accepted this patch, so if there is a more recent version of bcc somewhere someday, the patch should have been included...

Portability note: as86 makes a lot of presomptuous assumptions about type sizes, which prevents it from correctly running on architectures that fail to meet those. Hence, cross compiling, or even compiling from DOS, might be problematic, and will require patching some files (particularly see ld/typeconv.c about that problem). It should work on all 32-bit unixish architectures, as well as minix and Linux/ELKS.

Send patches to Bruce Evans (bde@zeta.org.au) and me (rideau@ens.fr)

Note for DOS users (): bcc is known to have been successfully used under DOS. I personally tried the following:

- To compile it under DOS you might have to define `POSIX_HEADERS_MISSING`.
- For bcc/as, if you're not using DJGPP, you will have to rename a variable named `far` in function mcall() in file mops.c, because some DOS compilers think `far` is a reserved keyword. Note that you need link with typeconf.obj from the bcc/ld directory...
- For bcc/ld, you'll need have a copy of a.out.h and ar.h; DJGPP has them, but other C compilers may require that you steal them from any GCC (under DOS, Linux, VSTa, etc).
- For bcc/ld, you need define the BSD_A_OUT macro for all files, and edit writebin.c so it define STANDARD_GNU_A_OUT and includes your copy of linux a.out.h with a usable DOS name.
- Linux a.out in turns includes asm/a.out.h, so you must manage to get it included, too. 16-bit compilers require that you edit the asm/a.out.h to modify a 24-bit bitfield into an equivalently sized set of bitfields lesser than 24 bits (ld doesn't use that field).
- I didn't try cc1, but it should be quite doable to compile it, too, if you like it; however, you'll have to rewrite part of the bcc frontend, or use cc1 directly, because it relies on the fork()/exec()/wait() treble to launch cc1, as, and ld when compiling.
- However, only the 32-bit-compiled version works fully correctly: the 16-bit one will output bad binary code, even though the listing from option -l looks fine!!!
- If you would recompile them with free compilers and send me, I'd appreciate a lot.

3.5.2 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.

3.5.3 Where to find docs

The docs are what is included in the bcc package. Man pages are also available somewhere on the FreeBSD site. When in doubt, the sources themselves are often a good docs: it's not very well commented, but the programming style is clear. You might try to see how as86 is used in Tunes 0.0.0.25...

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

Linus is buried alive in mail, and my patch for compiling Linux with a Linux a.out as86 didn't make it to him (!). Now, this shouldn't matter: just keep your as86 from the bin86 package in /usr/bin, and put 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.

3.6 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 useful if the assembler is part of what you're designing (i.e. an OS or development environment).

3.6.1 Win32Forth assembler

Win32Forth is a *free* 32-bit FORTH system that successfully runs under Win32s, Win95, Win/NT. It includes a free 32-bit assembler (either prefix or postfix syntax) integrated to the assembler. 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 – you could add that yourself, of course). Find it at ftp://ftp.forth.org/pub/Forth/win32for/

3.6.2 Terse

Terse is a programming tool that provides *THE* most compact assembler syntax for the intel x86 family! See http://www.terse.com

3.6.3 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 FAQ for comp.lang.asm.x86 http://www2.dgsys.com/˜raymoon/faq/asmfaq.zip

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 defined once, and reused multiple times, by automatically inline expansion, which allows safer programming, easier modification, etc. A "plain" assembler is often not enough, even when one is doing only small routines to link with C.

4.1 What's integrated into the above

4.1.1 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.

Then, you can make put your assembly into CPP macros, so anyone can use it in as any C function/macro. and inline C functions Inline functions resemble macros very much, but are sometimes cleaner to use. Beware that in 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. Also, note that some bug in your code or in GCC may appear when inlining functions with asm code where the register constraints weren't declared properly and/or confuse GCC.

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.

Beware that some optimizations that involve passing arguments to functions through registers may make those functions unsuitable to be called from assembly in the standard way, least you give them the attribute asmlinkage. See the linux kernel sources for examples.

4.1.2 GAS

GAS has absolutely NO macro capability included. However, GCC and passes .S files through CPP before to feed them to GAS. Files whose name ends in `.s` are the generated ones, and are passed directly to GAS. Again and again, see Linux sources for examples.

4.1.3 GASP

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

4.1.4 NASM

Some limited macro support (already?). If you have some bright idea, you might wanna contact the authors, as they are actively developing it. Meanwhile, see about external filters below.

4.1.5 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 in section 3.2)

4.1.6 OTHER ASSEMBLERS

Win32FORTH: CODE and END-CODE are macros that do not switch from interpretation mode, so you have access to the full power of FORTH words, immediate or not, 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) < \$< > \$@
```

4.2.1 CPP

CPP is truely 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 can't co-declared data and the code to process it, etc.

CPP came with your C compiler. If you could make it without one, don't bother fetching any (though I wonder how you could). GCC (see above) is a free C compiler you could have fetched.

4.2.2 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 from ftp://ftp.forth.org/pub/Forth/ in Reviewed/ ANS/ (?), or the Tunes 0.0.0.25 sources as examples of advanced macroprogramming using m4.

However, its fucked up quoting 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.

4.2.3 Macroprocessing with yer 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.

4.2.4 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 outputing 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, (particularly useful when some kind of data must be mirrored at into many cross-referencing tables and code chunks). etc.

Think about it !

Backends from existing compilers Compilers like SML/NJ, Objective CAML, MIT-Scheme, etc, do have their own generic assembler backend, which you might or not want to use, if you intend to generate code semi-automatically from the according languages.

The New-Jersey Machine-Code Toolkit There is a project, using the programming language Icon, to build a basis for producing assembly-manipulating code. See around
http://www.cs.virginia.edu/˜nr/toolkit/

Tunes The Tunes OS project is developping 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 symbolic 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/metaprograming.

http://www.eleves.ens.fr:8080/home/rideau/Tunes/

5 CALLING CONVENTIONS

5.1 Linux

5.1.1 Linking to GCC

That's the preferred way.

32-bit arguments are pushed down stack in reverse order (hence accessed/popped in the right order) above the 32-bit near return address. `%ebp`, `%esi`, `%edi`, `%ebx` are preserved, `%eax` holds 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 callee-save.

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` attribute for a function that will follow standard GCC calling conventions (I don't know what it does with modified calling conventions). See in the GCC info pages the section: `C Extensions::Extended Asm::`

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 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({\tt "}bar{\tt "}) (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.

5.1.3 Direct Linux syscalls

This is specifically *NOT* recommended, because it may change, it's not portable, it's a burden to write, it's redundant with the libc effort, AND it precludes fixes and extensions that are made to the libc, like, for instance the `zlibc` package, that does on-the-fly transparent decompression of gzip-compressed files. The standard, recommended way to call Linux system services is, and will stay, to go through the libc.

Shared objects should keep your stuff small. And if you really want smaller binaries, do use `#!` stuff, with the interpreter having all the overhead you want to keep out of your binaries.

Now, if for some reason, you don't want to link to the libc, go get the libc and understand how it works! After all, you're pretending to replace it, ain't you? You might see how linux-eforth-1.0c.tgz does it ftp://ftp.forth.org/pub/Forth/Linux/ The sources for Linux come in handy, too, particularly the asm/unistd.h header file, that describes how to do system calls...

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.

5.1.4 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 the GGI project: http://synergy.caltech.edu/~ggi/ http://sunserver1.rz.uni-duesseldorf.de/~becka/doc/scrdrv.html

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

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 were in real mode (I doubt they have stubs to have things work with 32-bit operands by calling 16-bit DOS services as needed).

Docs about DPMI and such (and much more) can be found on ftp://x2ftp.oulu.fi/pub/msdos/programming/

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 sunsite.unc.edu Also see the MOSS dos-extender from the Flux project in utah.

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

5.3 Winblows and suches

Hey, this document covers only free software. Ring me when Winblows becomes free, or when there are free dev tools for it!

Well, there might be after all; I've heard about cygwin32 from Cygnus support.

5.4 Yer very own OS

That′swhatmanyasmprogrammerstalkabout

5.4.1 Boot loader code & getting into 32-bit mode

5.4.2 The basics about protection

5.4.3 Handling Interrupts

5.4.4 V86/R86 mode for using 16-bit system services.

5.4.5 Defining your object format and calling conventions

5.4.6 Where to find info about it all.

Pleaseaddpointerstootherdocumentstothissection

The main source for information is sources of existing OSes. Lots of pointers lie in the following WWW page: http://www.eleves.ens.fr:8080/home/rideau/Tunes/Review/OSes.html

Particularly interesting are Cygnus support's ftp.cygnus.com archive (or a mirror like ftp://sunsite.doc.ic.ac.uk/packages/gnu/cygnus/), and the Flux project (http://ww.cs.utah.edu/projects/flux/).

6 TODO & POINTERS

- fill incomplete sections
- add more pointers to software...
- add simple examples from real life to illustrate the syntax, power, and limitations of each proposed solution.
- ask people to help with this HOWTO
- find someone who has got some time to takeover the maintenance
- perhaps give a few words for assembly on other platforms?
- A few pointers
  - *PM FAQ* `<ftp://zfja-gate.fuw.edu.pl/cpu/protect.mod>`
  - http://www.fys.ruu.nl/˜faber/Amain.html
  - http://alaska.net/˜rrose/assembly.htm
  - http://www.cera.com
  - http://www.cit.ac.nz/smac/csware.htm

  - *game programming* <http://www.ee.ucl.ac.uk/~phart/gameprog.html>

- And of course, do use your usual Internet Search Tools to look for more information, and tell me anything interesting you find!

Authors' .sig:

```
--    ,                                         ,           \_{} v     ~  ^  --
-- Fare -- rideau@clipper.ens.fr -- Francois-Rene Rideau -- +)ang-Vu Ban --
--                                      '                   / .           --
Join the TUNES project for a computing system based on computing freedom !
                 TUNES is a Useful, Not Expedient System
WWW page at URL: http://www.eleves.ens.fr:8080/home/rideau/Tunes/
```