# Win32ASM

## Disclaimer

This documentation and associated files is provided "as is" and any express or implied warranties, including, but not limited to, the implied warranties of merchantibility and fitness for a particular purpose are disclaimed. In no event shall the author be liable for any direct, indirect, incidental, special, exemplary, or consequential damages (including, but not limited to, procurement of substitute goods or services; loss of use, data, or profits; or business interruption) however caused and on any theory of liability, whether in contract, strict liability, or tort (including negligence or otherwise) arising in any way out of the use of this software, even if advised of the possibility of such damage.

## Distribution

Since this documentation and associated files are declared as Public Domain, you are allowed to distribute it without any restrictions on any storage or communications media, as long as you use the distribution self-extracting file without any modification, using the same file name (Win32ASM) as the original file.

## Trademarks

All brand names and product names used in this documentation are trademarks, registered trademarks, or trade names of their respective holders.

# 1 Introduction

Microsoft never documented the way to develop applications for Win32 Intel platforms using assembly language. The only assembly language documentation that Microsoft ever produced on the topic is the Win95 DDK, dedicated to the development of virtual drivers (VxD) in the Win95 environment, and it scarcely covers any Ring 3 application programming matter.
In addition; development in a Win32 environment requires the use of numerous reference data, such as function prototypes; structures, type and constant definitions, macros and other data. Microsoft released these items as C header files (.H files) in the Win32 SDK, but no equivalent files were ever published for assembly language.
This complete lack of documentation and tools propagates the illusion that developing in assembly language for Win32 is something that simply could not be done. This is aggravated by answers commonly provided by Microsoft's developer's support staff, claiming that "assembly language programming for Win32 is not supported by Microsoft" and even more often that "No, it cannot be done."

The truth is very different. Programming in assembly language for Win32 is indeed very possible,

- it is just as simple to achieve as with a High Level Language (HLL),
- there is nothing magic about it,
- the tools are the same, and
- with some initial explanations and road-mapping, the considerable C-oriented documentation that Microsoft (and others) have released can be used to program in assembly language.

Moreover, programming in the Win32 environment is paradoxically ***considerably easier now than it has ever been:***

- The Win32 API provides a vastly improved equivalent to the standard runtime library assembly language programmers never had before,
- the new operating environment, with its true multi-tasking services, provides a new context and new challenges for high performance assembly language applications,
- the Win32 environment offers the assembly language programmer additional debugging aids, tools and protection that never existed before.

Two critical keys are missing today to make the above facts obvious: Assembly specific documentation, and include files describing various symbols such as function prototypes, typedefs, structures and constant definitions.

This quick documentation tries to fill a part of the first shortcoming, and we hope the accompanying set of include files will be one step in the right direction toward remedying the second.

The bottom line is:
Programming in assembly language for the Intel PC platform has ***never been easier*** than it is today in the Win32 environment, and we hope this document will help you take advantage of the new opportunities this opens.

## 1.1 What this document is not about

- This document is ***not*** a tutorial on assembly language programming

- This document is ***not*** a tutorial about the Win32 API or Win32 programming issues. We realize that there is a need for initiation textbooks on the topic, but we believe that the first step to accomplish is simply to show that assembly language programming on Win32 platforms is ***easy***. Once this point becomes more obvious, we hope that other individuals will start developing new material (or adapting from existing bases).

- This document does ***not*** discuss the relevance of assembly language programming in today's world, nor the compared benefits and drawbacks of assembly language programming vs. HLL language programming. We assume that anyone reading this document has the intellectual ability to decide
  - whether assembly language is relevant, useful, fun, efficient, cost effective and/or needed, either generally or in any specific case, and
  - whether the much touted benefits of platform independence are relevant to one's particular situation, application and marketing strategies.

- This document does ***not*** reproduce information that can be found in the Microsoft documentation. Although we realize that it would be easier for the reader, it creates at least three problems:
  - The Microsoft material is copyrighted
  - The material to copy and/or paraphrase would be huge,
  - As time and new releases go by, the Microsoft material is frequently updated, and we simply could not keep up with it.

  For these reasons, whenever relevant documentation already exists, this document will only provide "pointers" to the Microsoft documentation (or to any other documentation, for that matter).

- Finally, this document is ***not*** be considered as a finished product, nor are the accompanying files. This only reflects an ongoing work, subject to changes, corrections and (many) potential improvements. **Contributions are welcome.**

## 1.2 What this document is about (and prerequisites)

This document explains ways to develop, link and debug arbitrary large assembly language projects in (and for) an Intel WIntel32 environment (*i.e.* Win95 or NT for Intel platforms). It describes various tricks, tips and recipes that we collected the hard way, by trial and error, documentation reading, debugger abuse and various other techniques.

It is intended for assembly language programmers

- who are already fluent in Intel (32-bit) assembly language,
- who know enough Win32 programming to write (or at least read and fully understand) a Win32 program in the C language,
- who have access to and understanding of the Microsoft's Win32 SDK documentation and tools and
- who are looking for all the useful details that Microsoft carefully refrained from explicitly documenting.

The "official" information about Win32 programming can be found in **M**icrosoft **D**eveloper's **N**etwork (***MSDN***). The Win32 SDK is delivered with MSDN Level 2 (a.k.a. "Professional") and upper level subscriptions.
Building applications using the techniques indicated in this documentation requires access to Microsoft's Win32SDK documentation and Microsoft's Win32 import library (.LIB) files. Both are available as part of Microsoft's Microsoft Developer's Network (MSDN). Both are also distributed with various 32-bit compiler sets, but as this document assumes and documents the use of the Microsoft Assembler, you must insure that the import .LIB you are using are somehow compatible with the regular Microsoft programming tools.

Talking about documentation, there is one public domain piece of documentation that anyone interested in Win32 programming should get:
In March 1996, Sven B. Schreiber released to the public domain a wonderful set of tools, with complete source and documentation. The whole set is available on the Internet at Sven's site, as file
ftp://ftp.orgon.com/pub/asm/WALK32_1.ZIP
It is also available from several other sources (try your favorite search engine and/or Archie program), and on Compuserve, GO PCPROG, Library 1 (Assembler)
I could not use Sven's tools, first because I found them too late and then because I needed Microsoft's COFF format compatibility and symbolic debugging support.
But I still wish I had found WALK32 earlier than I did. In addition to the collection of tools it provides, WALK32 includes remarkable documentation about Win32 and Win32 programming that can be used in any context. The documentation exposes many sound and clean programming techniques, including a few that I wish Microsoft had thought about when they originally designed the Windows API.
Whatever the tools you end up using, there are a lot of ideas, techniques and code that you can reuse from WALK32, and you shouldn't pass it.
Sven also published an article in the November 96 issue of Dr Dobb's Journal that shows a special application of WALK32 in Netware programming. The source code (including a mini WALK32 environment) can be found on www.ddj.com

Lots of other information can be found in any good computer book shop, as Win32 is a fashionable enough topic these days. But keep in mind that:

- Many books claim to cover Win32 programming; but not so many cover it properly.

- Many books are mere copying and/or rephrasing of Microsoft documentation and examples.
- No book so far really covers Win32 programming for assembly language. So assembly language programmers have to read at least another programming language in order to find the information they need about Win32. The “other language” that is needed is C. The official Win32 SDK by Microsoft describes most of the interface to Win32 in term of C language and C data structures. All function prototypes, structure descriptions and constant definitions are described in “.H” (C header) files in the MS SDK.
- There are many other books about Win32 and other languages (C++, Delphi and Visual Basic come to mind). We do not recommend attempting to use any of these books for our particular purpose, as the languages they cover offer a higher level of abstraction than C: As such, they tend to hide interfacing details away from the programmer and to make transposition to assembly language harder. It often becomes mostly impossible to relate the examples to the underlying machine (assembly) implementation. Be particularly careful when selecting new books about Win32 programming, as an increasing number of books cover the topic exclusively through C++ and MFC without ever mentioning this fact on the cover.

The short bibliography at the end of this document mentions a few reference books and magazines we found useful (and sometimes more) in discovering assembly language issues for Win32.
If we had to pick only ONE third party Win32 book, it would likely be ”Advanced Windows (Third edition)” [Richter 97.01]. This book clearly explains all the important mechanisms in Win32 (with C code examples), covers most differences between the Win95 and NT (including NT 4.0) implementations, and exposes many, many of the pitfalls and oddly documented aspects you ***have to know*** when programming for Win32. Be warned: This book is probably not for the beginning Windows programmer. Beginning Windows programmers might want to start with “Programming Windows 95” [Petzold 96.01].
Finally, Matt Pietrek’s “Windows 95 System Programming Secrets” [Pietrek 95.01] covers many aspects of Windows 95 internals and contains lots of invaluable information about many Win32 topics.

For those readers who own a Win32 compatible HLL compiler, another source of information could sometimes be the .ASM files that are delivered with the source of their runtime library. Some interesting pieces can be found there, possibly bringing information about some advanced topics like Structured Error Handling (SEH).

Last but not least, this document refers to a number of advanced (and sometimes not so well known) features of MASM 6.1x. It assumes that you have access to the ***MASM Programmer’s Guide***, either in its paper form or its electronic form. The electronic form is available in **MSDN Archive Edition**, “Product Documentation/ Languages/ Macro Assembler 6.1 (16-bit)”. Do not be fooled by the “16-bit” mention in the table of contents of the electronic documentation: this documentation is the image of the latest printed documentation and ***does*** handle the 32-bit features of MASM as well. It is very unfortunate that Microsoft decided to bury the MASM documentation in the archive CD-ROM rather than in the MSDN Library were it belongs, since MASM 6.1 is a dual, 16-bit and 32-bit product, and its 32-bit part is alive and well. It is specially inconsistent, now that Microsoft has brought MASM

back to the MSDN Universal CD-ROMs, that the MASM documentation has not been restored as well.
At the time of this writing, there is a section about MASM in the MSDN Library [Product Documentation \ Languages \ Macro Assembler 6.11 for Windows NT (32-bit)], but it unfortunately contains no more than a few release notes about MASM 6.11.

For those using the electronic documentation in MSDN Archive Edition, and since the electronic documentation does not provide table of contents and/or page numbers, we will attempt to reference the MASM documentation through its "hierarchical path", as describing the tree organization that appears using MSDN's INFOVIEW viewer.
References to the documentation will thus look such as in:
"Chapter 1 Understanding Global Concepts/ Language Components of MASM/ Statements".
Unless specified otherwise, all references will be to the Programmers Guide, in MSDN Archive Edition, "Product Documentation/ Languages/ Macro Assembler 6.1 (16-bit)"

At this time, we (unfortunately) do not know of any third party book that could be considered as an exhaustive (and, one would hope, improved) replacement for the MASM Programmer's Guide.

## 1.3 Keep the ball rolling…

This documentation can be improved.
If you feel that you hit a stumbling block in Win32 programming that you think is related to the use of assembly language, please Email me at the address below. If the topic falls indeed in the category we are trying to cover; and if we can provide a solution, we will add it to this documentation.
If you discover some interesting piece of information, if you work out some solution to a tough Win32 assembly-language problem or more generally feel you have something that could be useful to other Win32 assembly language programmers and that you would like to contribute, please Email it.

My own, single resources to improve the situation as we know it today are limited. But there are undoubtedly enough of us assembly language programmers around the world to create the missing pieces and make life easier to all.

Philippe Auphelle
philippe@irci.ihub.com

# 2 Product choices

## 2.1 Choosing an assembler

### 2.1.1 MASM 6.11a, 6.11d and 6.12

### 2.1.1.1 MASM availability (and uncertain future)

At the time of this writing, and to the best of our knowledge, MASM 6.11a is the latest commercially available incarnation of Microsoft Macro Assembler. MASM. MASM 6.11a is not the ***latest*** release of the software, though: it can be patched to 6.11c, 6.11d and 6.12 (see below).
Until recently (September/ October 1997), one could question Microsoft's willingness to keep on supporting MASM:

- MASM did not appear ***anywhere*** in Microsoft's Web site, not even in the developer product list.
- MASM did not appear as a product in any of the MSDN Universal CD-ROM disks, although MSDN Universal contains by definition all of the current Microsoft development products.
- The documentation for MASM only appears in the MSDN Archive edition, "Product Documentation/ Languages/ Macro Assembler 6.1 (16-bit)." Since the MSDN Archive CD-ROM is dedicated to obsolete products, one can question the actual position of MASM in the Microsoft product line.
- Visual Studio (aka Developer Studio), Microsoft's universal IDE (Integrated Development Environment), has provision for supporting nearly all Microsoft translators. "Nearly", that is, except for MASM, and even though the current version of MASM supports just everything it needs to in term of IDE prerequisite functions: Debugging, local / global symbol handling, COFF code format, etc… (more about this later)

Then by the end of August 1997, several things happened:

Microsoft posted a patch on the Microsoft Web Site. This patch turns any MASM 6.11, 6.11a, or 6.11d. to the brand new MASM 6.12. The patch is available at:
http://support.microsoft.com/support/kb/articles/Q173/1/68.asp
At about the same time, MASM 6.11a was rehabilitated as a Microsoft product, as can now be seen at
http://www.microsoft.com/products/developer.htm
http://www.microsoft.com/products/prodref/450_ov.htm

Finally, MASM 6.1 was (at last!) included in the MSDN Universal Edition (Level 4), starting with the October 97 issue. The CD-ROM version also contains the patches required to build a MASM 6.12 image.

So at the time of this writing, MASM 6.12 is the latest version of Microsoft assembler.
According to the README.TXT file delivered with the 6.12 patch, 6.12 corrects a number of the bugs that plagued 6.11, and brings Pentium Pro (a.k.a. 686) and MMX instruction set support to MASM.

Most of this document was written during the MASM 6.11d area, so a few of the bugs and limitations that we mention here *might* have been fixed in 6.12. Likewise, new bugs that could have been implemented in MASM 6.12 are not covered yet.

### 2.1.1.2 MASM capabilities

MASM 6.1x contains a number of advanced features that are very useful to the Win32 programmer. Several of these capabilities were actually introduced with 6.0 and 6.1 (but not all of them worked properly back then). Among these features are:

- Full Win32 support. MASM 6.1x is a true Win32 console application, and supports long file names for source, object, listing and symbol files.
- Support of both objects formats, Intel OMF ("old") and COFF (Win32).
- Support of function prototyping (headers), as well as assembly time parameter checking.
- HLL interfacing features, with automatic handling of parameters, both at call time and inside the called function. There is even TYPEDEF support.
- Support of several ways of handling external definition, including support of "soft" externals through the EXTERNDEF directive (users of the late and missed OPTASM top assembler will appreciate this one).
- Generation of "decorated" names allowing link-time checking of the number of parameters passed to a function. This can be used with Win32 import libraries to prevent parameter list errors from crashing processes
- Support of local (stack) variables symbolically, including at debug time.
- Support of structured programming directives. These brings the capacity of building "GOTO less" (JMP less would be more appropriate here) programs. Traditional labels and jump instructions are of course still available and can be used to construct assembly language programs in the great tradition of assembly spaghetti style, that gave assembly language maintenance its incomparable reputation.
- Support of the INCLUDELIB directive to simplify the linking process and allow the automatic invocation of import libraries by the linker.
- Last but not least; MASM is now a very fast assembler when running under Win95 or NT. Providing the Win32 system has enough memory, its assembly speed doesn't significantly suffer from the compiling of very large and numerous include files and/or complex macros. When initiated through my editor, a typical source with a few includes and a number of complex macros assembles in a few seconds on my relatively slow Pentium 60. This includes MASM load time, and the fact that listing file generation is always turned on in my configuration.

.

MASM is far from being perfect, though. It would certainly require a few improvements in some areas (more on this later). But it is a very good tool as it is.

### 2.1.2 TASM 5.0

TASM is Borland's assembly language.
TASM 4.x already supported 32-bit programming (it required a patch to run in a Win95 DOS box, though).

When Borland announced the brand new TASM 5.0 in March 96, I rushed my order, thinking it would solve most of the remaining problems I had with MASM (like TASM had done in the past). Unfortunately, it turned out to be different: TASM, that once used to be more compatible with MASM than MASM itself, now fails to fully support a number of the new MASM capabilities that appeared with MASM 6.10. Several of these being features that make programming to Win32 much easier, I regretfully had to go the MASM way rather than the Borland one.

### 2.1.3 Other assemblers

To the best of our knowledge, there is today no other 32-bit assembler that is suitable to full Win32 support (debugging code included), and that has a syntax compatible with that of MASM.
The best Microsoft compatible assembler ever, OPTASM from SLR systems, unfortunately disappeared without ever reaching 32-bit state.
WATCOM has an assembler, but that is too far from MASM compatibility to be really useful out of the WATCOM world.
There is also a GNU assembler, but the exotic GNU syntax (AT&T) is way too far from the original Intel/Microsoft syntax commonly supported in the Wintel world to fit my integration needs.

## 2.2 Choosing a linker

The choice of the Microsoft assembler and the debugging formats it supports induces the choice of a Codeview compatible linker. The choice of the COFF format reduces the choice yet a bit more.
The Microsoft linker fits the bill. I successively used :

| | |
|---|---|
| Link32.exe 1.00 | Probably came from the NT SDK/DDK |
| Link.exe 3.00.5270 | From VC++ 4.0 |
| Link.exe 4.20.6164 | From VC++ 4.2 |
| Link.exe 5.00.7022 | From VC++5.0 |

The WATCOM linker might work too, but since the Microsoft linker is widely available and did what I expected, there was little incentive to look for something else, and I didn't check it further. At the time of this writing, other linkers I know of either don't support the Codeview debugging format (Borland) or only support OMF object format (Symantec, Borland).
Those deciding to favor the OMF object format might want to consider the excellent, feature rich and superfast OPTLINK linker: As far as I know, it is not sold anymore as a stand alone product but now belongs to the Symantec C++ development suite.

## 2.3 Choosing a debugger

There are three debuggers I know of that offer full symbolic debugging compatible with the MASM / Microsoft Link combination:

- Microsoft own Developer Studio GUI debugger,
- WATCOM's GUI debugger, sold with their WATCOM C++ package. I used the one sold with 10.6 for a while and it worked quite well. Mostly comparable to the Microsoft debugger, although generally faster to load and operate. I haven't upgraded my WATCOM compiler to version 11 yet, so I can't tell whether anything has changed in the latest version.
- Numega's SoftIce 3.x (or upper) debugger. This one sits in a class of its own. It supports full symbolic debugging like the two previous ones, has an order of magnitude more capabilities than any other Win32 debugger and allows tracing inside the OS, access to the OS internal structures and much more. It is not a GUI debugger: it will either switch the screen to character mode when the debugger is entered (preserving the GUI screen), use an alternate screen (off a second video adapter, either monochrome or VGA) or allow remote control through an other machine.

## 2.4 Choosing a GUI IDE

By the current definition (and available products), a GUI IDE (Integrated Development Environment) is a piece of software that consistently puts together an editor, a "make" facility, a linker, at least one language translator and a debugger.
The choice of an IDE ended up being all too simple:

### 2.4.1 The bad news is…

I currently know of no "full" IDE that can directly and naturally home MASM development under Win32 (I'd ***love*** to be proved wrong on this one!)
One could expect that Microsoft's overhyped Visual (Developer) Studio would have the minimal support required to integrate Microsoft's own MASM. But unfortunately, as I mentioned earlier, at the time of this writing, it doesn't.
Obviously, the question was asked often enough to bore to tears the folks at Microsoft's developers support. So they published a Knowledge Base article on the topic:

**Using the Development Studio or Visual Workbench with MASM**
**Article ID: Q106399**
**Revision Date: 07-DEC-1995**

The article presents four pitiful kludges requiring the user to do most of the work by hand for each project to be managed through the IDE. But unfortunately, Microsoft didn't bite the bullet and incorporate MASM inside Developer (Visual) Studio. Someone at Microsoft has still to realize that the whole purpose of the IDE is to allow the programmer to merely click or drag'n'drop source files in the module tree, not to impose further configuration chores.

### 2.4.2 The good news is…

With quite a bit of work, it is possible to assemble a set of tools that has many of the characteristics of a full-fledged IDE. Once the initial integration is done, creating new projects and managing existing ones is just as simple as one would expect using a specially designed IDE.

### 2.4.2.1 Programmer's IDE for Windows 95/NT v2.3

First, here is a site that I recently found and that everyone should know about:
http://www.execpc.com/~sbd/

This page, “Software By Design”, belongs to Gregory Braun. Gregory Braun wrote a whole collection of small and beautiful freeware utilities, most of them in both 16 and 32-bit versions.
Among these, PMAN32.ZIP, a “Programmer's IDE for Windows 95/NT v2.3”, a very simple but nicely crafted general purpose programmer’s IDE, designed to synchronize a command line code translator such as MASM with an editor, a debugger, a linker and a make program. It looks like PMAN32 needs Link and Nmake (because it generates directives files for these two utilities).
PMAN allows one to configure a “project”, defining the compiling and linking parameters. PMAN holds the list of modules and generates the Nmake file and link parameter file. PMAN relies on file associations to let you switch from one file to the next by clicking on modules in its file list. Well, just try it: This is a no risk offer, it’s an 80K download.
Gregory Braun also developed a free hex editor, a free toolbar, a free notebook, a free CardFile-like phonebook / dialer, and even a game, BattleStar. Last time I checked, there were 18 different nice, well-behaved programs, all under 100K, requiring no setup, runtime, bulky DLL and or complex installation. And all the ones I tried worked great.

### 2.4.2.2 Watcom’s 10.6 IDE.

The Watcom10.6 C/C++ compiler comes with an IDE.
The Watcom10.6 IDE is not as flashy as the Borland and Microsoft ones. But it contains nearly everything that’s needed for the job.
“Nearly” only, because the way one configures the Watcom IDE is by modifying a set of text files (.CFG extension), and none of the syntax of those files is officially documented. The configuration syntax is quite complex, the configuration files are large and customization is quite painful.
In other words, Watcom created their IDE as a customizable tool but unfortunately, did not intend to go as far as making it a fully open, tool independent program.
But with a some imagination and the help of a user-contributed file found on the Watcom CompuServe forum and documenting part of the .CFG files, it was possible to tailor the WatcomIDE to make it use MASM, LINK and all their options instead of native Watcom tools.
Once this had been done, replacing the support for the Watcom debugger with that of the Developer Studio or SoftICE was not such a complex matter.

The benefits are:

- The resulting IDE uses its native WATCOM make program and provides a graphical interface to it (no fiddling with yet another disgusting "make" syntax).
- After proper configuration, The Watcom IDE gives access to all of the options for all tools, (MASM, LINK,… ) as radio buttons, checkboxes, edit boxes, etc… These are maintained individually on a file by file basis and can be changed through the GUI interface.
- Finally, the Watcom IDE supports multiple targets and cascaded dependencies, like a set of libraries and several .EXE files. If you modify one of the source files for a library, and this library is used to link an .EXE files, rebuilding the .EXE will first automatically rebuild the library (and recompile the source file). Unfortunately, there does not seem to be support for non-Watcom include files.

WATCOM supplies a text editor, but I didn't like it too much. I personally use American Cybernetics' Multi Edit for Windows (MEW). It has off-the-shelf support for all the language translators I ever heard of, plus many more I never thought could even exist. It does in-editor compiles either synchronously or asynchronously. It has myriads of customization capabilities, all accessible through nice hierarchical menus. And if that were not enough, it has a full macro language, and macros are provided with sources so you can change them to your will (I actually hardly ever needed this).
Last but not least, MEW has built-in support for several IDEs from various vendors, including the WATCOM one. Practically, this means that clicking on an error line reported by MASM to the WATCOM IDE brings up the MEW editor with its cursor pointing to the right line number. Pressing the "build" button in the IDE automatically directs the editor to save the source files before the IDE launches its "make" session. Etc…
Last time I checked, you could get an evaluation copy of MEW 7.1x from www.amcyber.com. And ***NO***, they don't bribe me for plugs!

Alternately, here is freeware bargain:
There are dozens of text editors available on the Internet, as substitute for NotePad and/or WordPad. But there are not so many that are good programmer's editors.

Here is the best we found:
"Programmer's File Editor" (PFE32), written by Alan Phillips. PFE32 is a full-fledged programmer's editor and supports a compiler / assembler, keyboard macros and other features. Last but not least, there is even an Alpha and a PowerPC version.

PFE32 can be downloaded from
http://www.lancs.ac.uk/people/cpaap/pfe
and many other sites (***hint:*** use an Archie search).
At the time I'm writing this, the name of the file is PFE0701I.ZIP, but this is likely to change as new versions are published. Go to the URL above if in doubt. The author can also be reached at
A.Phillips@lancaster.ac.uk

It was a lot of boring work to put all the IDE pieces together (specially customizing the Watcom IDE), but retrospectively, the result was probably worth the effort. I just wish one day, the good people at American Cybernetics bit the bullet and added to their great editor a fully configurable GUI "make", as well as the minimal additional support required to interface with the most common linkers and debuggers. This would turn a great editor into the first ***fast***, compiler-independent GUI IDE, something the development world is missing.

If anyone out there has found any useful, fully GUI combination of tools covering the same (or a larger) area, I'd love to hear about it!

# 3 Building an assembly language Win32 application

## 3.1 Using MASM

### 3.1.1 MASM Vs ML

The latest incarnation of MASM is not MASM.EXE, as once was. The true name of MASM is now ML.EXE.
MASM.EXE still exists in Microsoft's Assembly language package, but only as a mere shell that provides command line compatibility to older versions of MASM.
Both the MASM and ML names are used in this text and always refer to the latest, greatest ML.EXE.

The latest version of ML is version 6.12.
MASM 6.11d is available on MSDN Level 2 (and higher levels), on the NT DDK CD-ROM, in the \DDK\BIN\I386\FREE directory (files ML.EXE & ML.ERR), and can be turned into 6.12 by a patch we mentioned above.
The previous version, 6.11c, is also available on the Win95 DDK CD-ROM, but I don't know of any valid reason to use it rather than 6.11d (or 6.12).

### 3.1.2 MASM Documentation

Anyone who wants to program in assembly language might want to buy the Microsoft MASM 6.1x package, whether or not one already owns a copy of ML.EXE through the MSDN CD-ROMs: The reason is the Programmer's Guide book included in the official MASM package. This book doesn't exist in any other current source (specifically, it does NOT exist in MSDN Library, unlike all other Microsoft development products documentation). Nor do we have any third party substitute, like a good reference book on assembly language covering all the features of ML.
The worst part of the story is that the MASM programmer's guide is truly a terrible piece of documentation: It contains many invaluable little pieces of information. But they have been carefully buried all over the book in weird and absurd places, like Easter eggs, apparently to make absolutely sure only people ***very*** serious about assembly would ***possibly*** find them. That is, if they ***really*** tried hard enough... And no other book but the MASM Programmer's guide documents these things... The other manuals in the pack are obsolete and mostly useless for Win32 programming, as are most other programs on the MASM 6.1x distribution diskettes.

Another thing you need to do to complete your quest for MASM documentation is to search the whole MSDN library CD for the string "6.11." This will pull various items that include READMEs and knowledge base articles, detailing a number of small improvements, limitations and features that are not mentioned anywhere else. Some of these knowledge base articles are either obsolete, or wrong, or both, but heck, you can't win all the time.

Finally, the README file included in the MASM 6.12 patch contains a number of documentation updates and clarifications (some of them identical to previously published Knowledge Base articles.)
Read them carefully.

## 3.1.3 Your MASM source code

### 3.1.3.1 Use of registers

Upon return from a Win32 function, the function return value, if any, can be found in EAX.
All other values are returned through variables passed in the function parameter list you defined for the call.

A Win32 function that you call will always preserve the segment registers and the EBX, EDI, ESI and EBP registers. Conversely, ECX and EDX are considered scratch registers and are always undefined upon return from a Win32 function. EAX is never preserved either, and as we saw above, it most of the times contains a return value. Otherwise, it is void.
This convention is derived from the 32-bit PASCAL convention for register use. A little known fact is also that a Pascal procedure expects the direction flag to be clear upon entry, and must keep it clear upon exit (MASM Programmer's Guide, page 311).

If your assembly language PROC uses any of these precious registers, you might have to preserve them too. The simplest way to do so with MASM is by using the USES clause of the PROC directive, as follows:

```
Foo     PROC USES EBX ESI   ;Proc only changes EBX and ESI internally.
```

MASM will generate the appropriate PUSHes upon entry and the required POPs automatically before each subsequent RET instruction in the PROC. Of course, since you are undoubtedly programming in a structured way, there will be a single exit point to your function, a single RET instruction, and this will never generate more than the minimum number of POPs.

EBP is a special case, that you might not find too often in the USES list. If your function uses any local data (aka "dynamic" data, defined on the stack through the LOCAL statement), or if your function is called with stack parameters (declared in the PROTO / PROC definition), then MASM will generate the appropriate stack frame. It will set EBP as its addressing base to access both local variable and parameters. In this situation, MASM will also automatically generate code to save and restore EBP, so it would be a waste to mention the EBP register in the USES list.
Be especially careful to NOT change EBP in any PROC that is defined as taking parameters and/or using local data – or at least, to use EBP very carefully in such cases. Using the command line switches /Fl /Sg creates an expanded listing file showing all generated instruction and helps in mastering these delicate situations.

The above rules about register preservation might or might not apply to your own functions, though.
The general rules are as follows:

Win32 functions that you call from your own code do not care about the entry contents of the EBX, EDI, ESI and EBP registers.
Win32 functions currently **do** care about the content of the segment registers, though, assume them to follow the holly FLAT model, and don't bother reloading them upon entry into system code. In other words, if you ever happen to play with segment registers, Win32 is very likely to expect the segment registers upon function entry to be in the same state as they were when your process initially got control.

When calling functions that ***you*** wrote from ***your own*** code, it is obviously your own business to decide whether the calling function, the called function or neither of them will save and restore registers. You might be able to spare quite a few cycles by only saving registers when you know it is really needed: it is not because MASM gives you some HLL-like facilities that you should start generating code like that of a compiler!
On the other hand, functions that you wrote and that you register as a ***callback*** with any Win32 or other function external to your code should ***always*** play by the rules: The upper level Win32 function that will eventually call your code certainly doesn't expect you to change any of EBX, EDI, ESI, EBP and the segment registers before returning control. So callback functions that you write should ***always*** return with these registers unaltered.
**The very same rule applies to the DLL functions you export**: the code calling your DLL expects your code to play by the rules and respect EBX, EDI, ESI and EBP.
A quick one about the segment registers: you will probably not need to do anything with the segment registers. As you certainly know, Win32 uses the FLAT model, where all segment registers contain descriptors mapping the whole logical address space your application (process) sees. The way the flat model was implemented by Microsoft is too restrictive in my opinion, and deprives the programmer of a very useful and efficient native CPU mechanism, as is explained below in "The absence of LDT support in Intel-based platforms", page 44.
But this is unfortunately the way it is today. Unless proved otherwise, it's a no-no to change DS, ES, CS, SS or FS. The GS register doesn't ***seem*** to be used, but to my knowledge, nothing has been published on the topic so far and I did not investigate any further yet personally. If you ever try to play with GS, you should at the very least realize that you are doing it at your own risks.

In any case, in the great tradition, although Microsoft didn't let the Ring 3 application programmer use segmentation and segment registers, they permitted themselves to use it, even in your own Ring 3 code:
If you look at the FS register, you'll notice that at any time, the FS register contains a valid selector. This has been documented in [Pietrek 95.01]. The selector in FS actually points to a Thread Information Block (TIB). The TIB contains various thread-dependant items. The contents of the TIB are used by many Win32 syscalls; and changing the FS register is very likely to crash your process automagically – and very soon.

### 3.1.3.2 Function call conventions

Interfacing with Win32 is designed for HLLs, and MASM makes provision for predefined “Function Call Conventions”, that let it take care of a few boring matters for you. MASM allows you to pick one from C, PASCAL, STDCALL, SYSCALL.
But Win32 always requires and only supports STDCALL, with one exception (explained below).

The function call convention is used in conjunction with the PROTO, PROC and INVOKES directives. You define a default for the whole module by using the .MODEL directive ahead of your source file, and this saves you from repeating the calling convention for each subsequent PROC or PROTO you define.

```
.386          ;(could be 486, 586,or 686 too)
.MODEL FLAT,STDCALL
```

Defining a given default calling convention in .MODEL still allows you to override the default for any given function. The calling convention you explicitly state in specific PROC and/or PROTO directives overrides the one in .MODEL

The calling convention defines two different aspects of function interfacing:

- Naming
- Parameter passing

STDCALL is an hybrid of the C convention, for the naming and order or parameters of the stack, and PASCAL convention, for the removal of parameters from the stack.

#### 3.1.3.2.1 The naming convention

The naming convention defines the way the names you assign to symbols in your modules appear outside of your module, i.e. at link time.
The STDCALL naming convention tells MASM to "decorate" subsequent PROC names, prefixing them with an underscore (_) and postfixing them with a @ sign, followed by the number of bytes in the parameter list (this is the type of decoration the Microsoft C compilers generate).
For instance, a function Foo that would take two DWORD parameters would have its name turned into the external (link time) name

```
_Foo@8
```

by the assembler (since two DWORD generate 8 bytes of parameters). This trick is used by the linker to perform some brute force (but ***very*** useful) parameter consistency check against the Win32 import libraries. If you inadvertently specified the wrong number of parameters in the PROC and PROTO definition, MASM would generate the wrong “@x” postfix value and the link would fail with an undefined reference, pointing at the error. Believe me, this is ***much*** better than getting some unpredictable behavior at runtime because of a parameter error, disguised into a stack error…

#### 3.1.3.2.2 The parameter passing convention

The parameter passing convention defines the way the parameters in the parameter list are pushed on the stack, and the way they are removed from the stack.
The STDCALL calling convention defines that parameters are pushed on the stack right to left (i.e. the last parameter of the list is pushed first). It also defines that the callee (rather than the caller) will remove the parameters from the stack.
Having the callee remove the parameters has two benefits:

- it saves an instruction for each call in the code (as the stack cleanup code does not need to be repeated in the code after each call)
- it is much more likely to trigger a trap (exception) if the caller calls using the wrong parameter list: as the callee will remove what should be the number of bytes in the parameter list, the resulting stack misalignment is likely to trigger an exception very soon. Using the C convention, the calling code would always consistently remove the number of bytes it placed on the stack, and the error could go unnoticed for a long time.

Like with any good rule, there is one exception to the use of STDCALL in Win32: functions accepting a variable number of arguments require that you define them as using the C calling convention. With such functions, the callee can't remove the parameters from the stack, since it doesn't know until runtime how many of them where pushed in the first place. So the caller does the cleanup. To the best of my knowledge, this exception only applies to ONE function in the whole, huge Win32 API: the ***wsprintf*** function. Other functions requiring a variable number of data items (like wvsprinter and FormatMessage) actually take a fixed number of parameters including a "va_list" parameter instead, i.e. a pointer to a list of arguments that ***can*** be variable.
The MASM Programmer's Guide claims (in Chapter12/ Naming and Calling Conventions/ The STDCALL and SYSCALL Calling Conventions, page 311) that prototyping an STDCALL function with the VARARG keyword achieves the same effect as declaring it C: But it appears that in fact, MASM flags this as a programmer error (*true in 6.11d, not checked in 6.12*).

Using PROTO definition and INVOKE statements, MASM will generate the right set of instructions to push (and for the C function, to remove) parameters from the stack.

I highly recommend that you use PROTO definitions to define external functions (whether imported or exported), and the INVOKE directive (rather than a series of PUSH followed by a CALL) to generate the boring list of PUSHes needed to call Win32 functions. There is a small limitation to the use of this directive (see below, "The infamous 512 bytes buffer, page 35"), but it can be dealt with.

PROTO definitions work in a way similar to function prototypes for C compilers: they define function headers that the assembler uses

- to generate the right calling sequence when the function is invoked but not defined in the module,
- to check parameter consistency in the module where the function is actually defined.

### 3.1.3.3 Win32 (and other) function prototypes

The best way to handle Win32 API calls is to create include files containing PROTO directives for the various Win32 functions, as well as the equates and structures they require.

After a while, we found out that the simplest way to organize this was to stick to the scheme used in the Win32 SDK: whenever possible,

- use an include file for each import library (or DLL) that is needed from the SDK, and
- give the include file the same name as the import .LIB (and DLL) with some distinctive extension name (like .EQU, .INC or HDR),
- When defining the PROTO headers, keep ***exactly*** the same parameter names as the ones used in the SDK. This is not mandatory, as parameter names are just placeholders in the PROTOs, but it will help you relating the SDK documentation with your ASM documentation
- When defining structures and equates, stick ***exactly*** to the spelling and case of the original equates, structure names and structure members. If you are using TYPEDEFs, do this for TYPEDEFs too. For equates, structures and structure members, this is a must, as these names are the ones you will find in the SDK documentation, examples and third party books. You don't want to spend half your development time looking up the nice original name you found for that error code or structure member, do you? There is one case when this rule cannot apply, and this is when a structure member collides with a MASM reserved word, like for instance, OFFSET. In this case, use a simple, consistent, automatic renaming convention to create a non-colliding name. AND clearly document the trick for posterity. I personally decided to prefix the name with an underscore ('_').

Following these simple rules makes it very easy to organize your include files. Lookup a function in the MSDN InfoView documentation; constants, return codes and structure names are the same as the ones mentioned in the SDK so you can use them as is. Now press the "QuickInfo" button, look at the name of the import library, et voilà, you directly know which EQUate (PROTO) file to include in your source file.

Both MASM and TASM come with utilities to convert C header files (.H files) to include files (.INC files). The Microsoft one is H2INC.EXE, while the TASM one is H2ASH32.EXE. I ended up not using them, because:

- neither is able to properly compile original unmodified .H files. Both spit tens of errors compiling WINBASE.H, for instance. And manually tweaking copies of the original .H files to get them to properly convert is a pain.
- I decided I preferred to know what I imported in my files rather than blindely importing whatever the converter converted. There are many things in the .H files that might have some historical value (for 16-bit portability, for instance), but do not mean a thing in a new Win32 assembly context.
- I didn't like the twisted syntax the Microsoft tool (H2INC) generates for function prototypes: a TYPEDEF with an artificial name (PROTO_<sequence number>) followed by a PROTO referencing the TYPEDEF.

As a result, I chose to create include files for each of the libraries I needed manually, and to add function prototypes, structures and equate definitions along the way, as I needed them.

The ideal solution to this problem could be an assembler that would properly compile and interpret a large subset of the standard .H files, smartly enough to gobble irrelevant errors. Hey, one can dream a little! On the dark side, this would likely slowdown assembly by a large amount.

Alternately, an H2INC-style utility that would be able to properly and cleanly compile any of the existing .H files to .ASM source would certainly help.

### 3.1.3.4 The INCLUDELIB directive

A very useful MASM feature is the INCLUDELIB MASM directive (MASM Programmer's Guide, "Chapter 8/ Developing Libraries/ Associating Libraries with Modules", p. 222): Insert an

```
INCLUDELIB KERNEL32.LIB
```

directive ahead of the include file that contains the PROTO definitions for the KERNEL32 functions, for instance, and MASM will automatically generates a linker directive (embedded in your object code) adding KERNEL32.LIB to the list of libraries to search at link time.

MS Link is able to recognize and process the embedded directive. I didn't check with other linkers, but I would assume they understand the embedded directive the same way.

By using this technique, the only other thing that the linker needs to resolve external references is a LIBPATH switch on the command line: A command-line switch like /LIBPATH:G:\WinSDK\LIB tells MS Link where the Win32 import libraries files (that you might define using INCLUDELIB)can be found. If you have several groups of libraries, use the /LIBPATH directive several times on the command line.

One of the MASM-related Knowledge Base articles claims that INCLUDELIB is not supported with LINK: Don't believe it, INCLUDELIB does work just great, at least with recent the 3 most recent MS LINK implementations I checked.

Additional tricks:

If you need to include several libraries, use several INCLUDELIB directive.

INCLUDELIB can be used to pass other directives to the linker. What INCLUDELIB does is to embed a "-defaultlib:" directive in the special ".drctve" pseudo-section (see Microsoft definition of the COFF format for more information on that special section).

The (dirty) kludge we use here is that INCLUDELIB passes ***everything*** that follows it "as is" to the linker.

So a line such as

```
INCLUDELIB Kernel32.lib –verbose
```

Will actually pass

```
-defaultlib:Kernel32.lib –verbose
```

as a additional parameter line to the linker…

The END directive acts about the same as the INCLUDELIB directive. It only generates an
-entry
parameter rather than a –defaultlib one.
One can regret that Microsoft did not make provision for a generic LINKDIRECTIVE verb instead of (or in addition to) creating the specialized INCLUDELIB and END directives.

### 3.1.3.5 Segments and sections

There are no more “ segments ” in Win32 world, because they have been replaced by “ sections. ” You can still define sections the old way, using SEGMENT directives. The best way, adequate in most application, is to use simplified directives as offered by MASM.
There are mostly 4 predefined section names that are useful in Win32 programming:

.CODE
.DATA
.DATA?
.CONST

.CODE, as the name implies, defines the .CODE section. Don’t you ever try to write to it!
.DATA defines the initialized data section.
.DATA? defines an uninitialized data section.
.CONST defines a read-only section for constants. Can’t be written to either.

This defines “pre-canned” sections. But you can generate any customized section with any name and attributes using the segment directive. Be aware that this grows the size of the resulting .EXE file, and that this also wastes some memory: Sections are usually aligned on 4K page boundaries, so needlessly creating many sections containing each a few bytes could potentially waste nearly 4K bytes per section, even if the actual negative impact might be somewhat reduced by the VM management logic.
There are a few cases where you might want or need to declare additional sections, though, and let the linker reorganize the data at link time. This is the case if for some reason, data that you declare scattered all around many modules needs to be consolidated / concatenated in the runtime program image. This could also be the case with statically allocated memory that you would want to manipulate at runtime in a special way through the virtual memory set of system calls (now see why the sections have to be page aligned?)

### 3.1.3.6 Alignment issues

Alignment issues are critical to performance in 32-bit systems. This is something you should ***never*** overlook when programming in 32-bit assembly language, as misalignment will go unnoticed but silently kill your performances.

It is particularly important to remember this point when coding byte strings and structures in data sections: Coding something like

```
        ALIGN DWORD                 ;This is equivalent to ALIGN 4.

MyString     BYTE 'FooBar',0
SomeDWORD    DWORD 0                ;Watch out! Not aligned!
```

might results in severly degraded performances.

The variable "someDWORD" and the following DWORDs are not properly aligned, and can considerably slow operations down if accessed very frequently.

There are several ways to handle this:

- Manually insert ALIGN DWORD directives after each non-DWORD directive,
- Grouping data items by size, and prefixing each group with an ALIGN <size> directive,
- Creating additional sections (possibly by defining macros to define a .DATABYTE and a .DATAWORD sections). This is a costly solution, though (sections are allocated with a 4K page granularity).

Likewise, when coding structure, always group data items in such a way that

- DWORDs are always aligned on a DWORD boundary
- Words are always aligned on a WORD boundary
- When using bytes, group them by 4, or by 2 with an adjacent word, etc…

Finally, do not forget to mention structure alignment in your structures, such as this:

```
Foo STRUCT DWORD                    ;Equivalent to STRUCT 4
DWFoo0       DWORD –1
BBar         BYTE 1
DWFoo1       DWORD –1               ;Padding will be added before
                                    ;DWFoo1 to achieve proper alignment.
Foo ENDS
```

The alignment rules for structures are documented in MASM Programmer's Guide, "Chapter 5/ Structures and Unions/ Declaring Structures and Unions/ Alignement Value and Offsets for Structures", page 119. One thing that I have not seen documented is that the alignment specification after the STRUCT keyword can be a type specifier directive (like the DWORD in the example above), instead of a number (1, 2 or 4).

Look at the code generated by the above structure: MASM will pad item BBar with zeroes to respect the alignment inside the structure.

Since VC++4 / VC++5, Visual C++ started to default to aligning structures on QWORD boundaries. There is no real hardware reason to follow this rule with the current machines, as the cache lines for the Intel 486 and Pentium processors are 16

bytes and 32 bytes, respectively. Other Intel recommendations suggest aligning data the following way:
WORD data should not cross a DWORD boundary,
DWORD data should be aligned on a DWORD boundary,
QWORD data (double precision reals) should be aligned on an 8-byte boundary.

So in our 32-bit Intel world, I currently see no real reason to align on QWORDs. I suspect this change is nothing more than Microsoft's anticipation of the use of 64-bit machines, where the performance hit might occur when the native word format (QWORD) alignment is not respected. On 64-bit machines, it is safe to assume that the C integer will be… 64 bits, and that by simply aligning structures to QWORDs, the alignment issues would be solved. Previous experience suggests that portability issues are not limited in any way to simple word size and alignment matters, but this is yet another story.
The exact architecture of the future Intel machines is not known at this time, anyway, and the only points that have been disclosed tend to indicate that they will use "dual mode" machines, able to execute either in 32-bit or in 64-bit mode.

Now, considering that:

- assembly language is not that portable anyway,
- we don't know what the 32-bit code performance will be like on these machines, but might assume that Intel will make their best to make it look excellent to ease the transition, and
- nobody expects the 32-bit machines ***and code*** to disappear overnight (heck, most of the world is still mostly running 16 bits DOS code!),

The bottom line is, "I am not sure this is worth bothering at this point."

A last word on alignment:
Sven B. Schreiber brought to my attention that alignment issues are not only a performance issue: they are also reliability issue. Since NT 3.51, some APIs will simply crash your process if you pass them parameters that are not aligned on DWORD boundaries. Sven makes a special mention of resource data such as BITMAP structures that ***must always*** be DWORD aligned.

### 3.1.3.7 END statement and Entry Point.

***Note: The following applies to 6.11d. This issue is documented as "fixed" in MASM 6.12 README.TXT file, but I have not checked it yet.***

If you're using MS Link, you are likely to discover that putting a label field in the END directive of your main program to specify the entry point does not seem to work:

```
END Start       ;Will seemingly be ignored by MS Link.
```

This behavior seems to be consistent with the Knowledge Base article

**“32-Bit Flat Memory Model MASM Code for Windows NT”, Article ID: Q94314, Revision Date: 23-JAN-1995.**
Do NOT believe this article, it is all wrong. There is a bug in the way the END directive is implemented, but it can be worked around easily.

Here is the scoop: MASM 6.11d does process the Start label in the END directive and generates an embedded /ENTRY directive in the object code. The only problem is that it does not generate the right label in the “/entry:” directive it writes to the .OBJ file.

If an STDCALL directive is in effect, and the entry procedure is Start, the "END Start" directive will generate an inline

```
-Entry:_Start"
```

directive.
Now, note that LINK will in turn decorate the name it gets in the “entry” parameter and internally change it at link time to "__Start", that it expects to be a PUBLIC in the object file.

Strike 2: There can't be neither _Start nor __Start externally defined in the MASM module, because Win32 requires the use of STDCALL. And STDCALL will change the "Start" in the source code into _Start@0.
So MASM should be consistent with itself (and with LINK…), and apply the default interface convention to the END directive, thus generating an inline

```
/Entry:Start@0
```

link directive for everything to work fine.
Since MASM is inconsistent, we have to fix the problem ourselves. This can be done by replacing the END directive with an ENTRY directive, defined by the following macro:

```
ENTRY MACRO EntryPoint:REQ
   LOCAL EntryPoint
    IF @Version GE 611
    ALIAS <_&EntryPoint&@0>=<&EntryPoint>
    ENDIF
   END &EntryPoint
   ENDM
```

With this macro used in your code in place of the END directive, the code line

```
  ENTRY Start
```

should work just fine, by instructing the linker to use the right label in replacement to the one it can’t find. See the explanation of the ALIAS directive below (Use of ALIAS, page 34) for more details on this magic.

### 3.1.4 MASM options

You have to use the /Cp (or at least the /Cx) MASM command line option.
/Cp forces all identifiers to be case sensitive. All Win32 symbols are case sensitive.
You also have the option of using /Cx instead of /Cp, but I wouldn’t recommend it: With /Cx, external and public symbols are case sensitive while others are not. This is

the kind of sloppiness that is confusing at best, and usually strikes back when you expect it the least: If you suddenly decide to reorganize your code modules (by breaking out a very large code file, for instance), symbols that used to be internal might need to become external. And this will suddenly reveal discrepancies in the case usage. Since code restructuring is already a delicate operation, you usually don’t want this extra problem to surface at that time.

You will likely want to use the /c command line parameter, too: this will prevent MASM from automatically attempting to launch LINK. For all but the smallest projects, you will want to control the launching of LINK by other means.

Use /COFF code generation in ML. Link only knows COFF natively and you won't need to undergo a COFF to OMF conversion. The call to the OMF-to-COFF converter is handled by LINK (by shelling out to an external conversion program), but this only slows down the linking process.

Use /Zi to get symbolic debugging information.
This should work with any debugger directly supporting the Microsoft’s Codeview debugging information. We successfully tested Microsoft’s MSDEV IDE debugger, Watcom’s 10.6 debugger as well as with SoftICE for Win95 version 3.x.

Using the /W3 option to turn maximum warning level on might be useful too.

You might want to use /Sc to get instruction timings in the listing file. Timings depends on the definition you gave of the target CPU (.386, .486, .586, etc…).
This facility is quite useful, but:

- Keep in mind that this only shows brute force timings, and does ***not*** take into account other factors like CPU pipes, AGI stalls, cache side effects, alignment effects, etc… See [Booth, 96.01] or [Intel, 95.01] for more details on timings and processor dependant optimization.
- MASM unfortunately doesn’t show timings (or cumulated timings) for the code it generates through structured programming directives and/or macros.

### 3.1.5 Miscellaneous OS and systems issues

### 3.1.5.1 Beware of the CLI

At least with Win95, beware of the CLI instruction.
CLI is a privileged instruction, and as such, should not be used when running protected mode. This also applies to several other instructions, such as of course STI and a handful of others. One would expect a process using it from ring 3 code to be trapped by an OS exception, and the offending process terminated.
Not so with Win95: One day, porting some code from the DOS world, I accidentally left in a CLI instruction (with no subsequent STI nor POPF to reset the Interrupt flag to its original value) in a Win32 program. This resulted in a bug that it took me a long while to figure out. The problem of course did not show up at the place where the CLI instruction was. And to make things worse, that code was not always called either. When it happened, though, the execution of the CLI from a thread simply messed up the inter-thread cooperation with no apparent cause, and the resulted was that the whole process seemed to internally hang… without freezing the rest of the system. At some point, I came to suspect a leftover CLI, ran a text search for CLI on the whole

pile of sources and bingo, found the offending line. Everything went back to normal after I removed it…

When I told this story to Sven, he quickly pointed out that this Win95 behavior was carefully documented in "Unauthorized Windows 95" [Schulman 94.01], pages 319-331. Schulman documents the effect of CLI in Win95 under various conditions. The most worrisome ones are when the CLI (with no matching STI) happens in a 16-bit regular DOS program running in Intel's CPU V86 mode.
The effect I saw happens when running under protected mode, and affects only the faulty process.

Under the same circumstances, NT would quite simply trap the process with a "Privileged Instruction" exception.

### 3.1.5.2 Beware of the STD

If you ever set the Direction flag through the STD instruction, do NOT forget to clear it before reentering the OS. In a CallBack or DLL, CLD before returning to the caller. In your own code, CLD before calling any OS function.
Windows 95 is likely to crash if you forget to CLD. It seems that the kernel entry points do not insure the state of the D flag before starting to execute, and that at least some portions of the kernel code and/or device drivers expect and assume the direction flag to be clear when they get control.
I have no idea whether this affects NT.
In any case, just play it safe and do not forget to CLD after you STDed.

## 3.1.6 Various MASM goodies

The following are a few little known MASM capabilities or obscure point of documentation that could easily be missed:

### 3.1.6.1 Data Types

The "old" data definition directives (DB, DW, DD) have been extended and superseded by newer, more precise ones (Don't gasp; the "old" ones are still valid):

BYTE, SBYTE, WORD, SWORD, DWORD, SDWORD, FWORD, QWORD, TBYTE, REAL4, REAL8, REAL10.

They are defined in MASM Programmer's Guide, "Chapter 4/ Declaring Integer variables/ Allocating Memory for Integer Variables", page 86, and their 16- and 32-bit C equivalents are defined "Chapter 12/ The MASM/High-Level-Language Interface/ The C/MASM Interface", page 315.

### 3.1.6.2 Base and Index

The same registers can be used as either base or index registers. Both uses are not equivalent, specially in term of performances (refer to documents on Intel CPUs optimization). In normal situations, the “natural” syntax will achieve what you want. The syntax to control the base address register you want is defined in MASM Programmer’s Guide, “Chapter 3/ Operands/ Indirect Memory Operands with 32-bit Registers/ Scaling factors (end of paragraph)”, page 70.

### 3.1.6.3 Structures and Unions

MASM gives the programmer the ability to define nested structures, unions, and combinations of both.
This is defined in MASM Programmer’s Guide, “Chapter 5/ Structures and Unions/ Declaring Structure and Union Variables”, page 121.
One thing that is not defined so well (no example) is how to declare an array of uninitialized structures.

Providing Item is defined as a structure,

```
Foo   Item 10 DUP ({})
```

the above will define an array, Foo, of 10 uninitialized structures of type Item.
There is more information about this topic in the README file for the MASM 6.12 patch (*c.f.* supra.)

### 3.1.6.4 Local directive

The LOCAL directive is quite a useful one.
Contrary to popular belief, it is not limited to declaring DWORDs, though: Even strings or arrays can be defined the following way:

```
LOCAL EXP_BUF[5+1]:BYTE
```

(Do not forget the alignment issues!)

This funny syntax is documented in the MASM Programmer’s Guide, but you are likely to miss it, unless you look very carefully at the code fragment at “Chapter 7/ Procedures/ Creating Local Variables Automatically”, page 192: the example to look at is the “aproc” procedure, and the way it defines a local array of words...
You can't initialize a local var. The whole LOCAL space is merely allocated all at once on the stack at runtime upon proc entry and you have to MOV any initial value there all by yourself (look at the code the LOCAL statement generates). But you can of course use symbolic addressing to do so, and the assembler will automatically generate the corresponding EBP-based addressing.

It is probably useful here to mention a little know but very important characteristic of the Win32 stack management:
Sven B. Schreiber brought to my attention the sequence of code that the C compilers generate to “probe” the stack when assigning large LOCAL blocks. “Large” is defined here as near or above 4K (a VM page). The problem seemed to be that if a

program would unexpectedly hit a stack page that had not been committed, the whole process would suddenly disappear with no warning or message. Not even a “Poof!” or a cloud of smoke… This did not seem very clear at that point, as the stack probe would not seem to accomplish more than what the “faulty” program would do, *i.e.* merely touch the uncommitted page. I finally found the whole story in Richter’s “Advanced Windows” [Richter 97.01], in the chapter titled “Using Virtual Memory in Your Own Applications”, subtitle “A Thread’s Stack.”

The problem is that the stack normally grows and shrinks “sequentially.” The OS tracks the current bottom of the stack by trapping accesses to the lowest page that is currently committed, the “*guard page.*” If an access is done in that page, the OS commits yet another page immediately below, and this new page becomes the guard page. Trouble occurs if an application allocates more than 4K (a page) of data at one time and manages to directly access one page ***below*** the guard page, effectively jumping over the guard page and defeating the stack growth logic. This should normally trigger an access violation, but it instead silently kills the process, just like a stack overflow would do.

This problem is taken care of by the “stack checking” logic of the compilers: when the compiler detects that a function is allocating more than 4K of local data, it generates code that “touches” the allocated data sequentially, from top to bottom, 4K at a time. Whenever the guard page is touched, and new page is committed 4K below it and the newly committed page becomes the guard page, and the compiler stack probe routine prevents a stack fault from occurring this way.

Once again, this is fully documented with all gory details in the aforementioned Richter book.

The bad news is that this case is not currently handled by the MASM prologue code generated in STDCALL functions.

But there is good news too:

- One is that this problem is limited to functions that allocate 4K or more of local variables at a time, that there are probably not so many such functions, and that it is easy for the programmer to track these manually and add a call to a “stack touch” loop in these functions.
- Another good news: MASM makes provision for fully customizing the prologue and epilogue code that are generated upon PROC entry and exit. This is the right place to write the stack touch logic. The epilogue code has access to everything it needs, such as the size of the LOCALs for the function, the function’s .MODEL, etc…The way to achieve this is documented in the MASM Programmer’s Guide, “Chapter 7/ Procedures/ Generating Prologue and Epilogue Code”, page 198. One of the include files provided with MASM, “PROLOGUE.INC”, gives an example of a customized prologue code in a 16-bit environment.

### 3.1.6.5 INVOKE through a function pointer

This is specially needed when the function takes parameters.

The way to do this requires a TYPEDEF and is defined in “Chapter 7/ Procedures/ Calling Procedures with Invoke/ Invoking Procedures Indirectly” on top of page 198 in the MASM Programmer’s Guide.

Without parameters, CALL to a DWORD variable can be used: it is more straightforward to code (no TYPEDEF, etc…), and supports forward references.

### 3.1.6.6 Global labels

Labels are normally local to PROCs and can't be made external:

```
Foo:    ;This label is local to the PROC where it is defined.
```

You can make it global by using the following notation:

```
Foo::   ;This label is now global for the whole module.
```

(MASM Programmer's Guide, "Chapter 8/ Declaring Symbols Public and External / Using EXTERNDEF/ (look at label "codelabel" in the example) page 215)

Remember that most local labels can be eliminated by the use of structured programming directives.

### 3.1.6.7 Structured programming directives

Look carefully at, and ***do*** use the ML (MASM) structured programming directions.
MASM
has .IF / .ELSE / .ELSEIF /.ENDIF, .WHILE / .ENDW(HILE) / .BREAK / .BREAK . IF / .CONTINUE / CONTINUE .IF / .REPEAT / .UNTIL (MASM Programmer's Guide, "Chapter 7/ Jumps/ Conditional Jumps/ Decision Directives", pages 171, up to and including "Chapter 7/ Loops/ Writing Loop Conditions/ Expression Evaluation", page 179).
These directives generate jumps and conditional jumps, with the exception of the .IF directive that can also generate logic instructions. The generated code is very easy to control, and as such, fully stays under programmer's control. The removal of extraneous labels and jumps from the source code makes the code much easier to read at virtually no efficiency cost. It also relieves the programmer from having to find zillions of meaningful labels.
The list of expression operators accepted by the conditional directives (for both .IF and loop directives) is defined at "Chapter 7/ Loops/ Writing Loop Conditions/ Expression Operators", page 178.
Expressions are normally unsigned, but they can be forced to be signed. The way to accomplish this is defined in the paragraph starting at "Chapter 7/ Loops/ Writing Loop Conditions/ Signed and Unsigned Operands", page 178.
The only shortcoming is that there are some limitations in the .IF capability of testing for preexisting several complex combinations of condition codes that have JMP equivalent, like BE (below or equal), etc. But since most of the times, the .IF instruction generates the condition codes itself, the problem seldom occurs.

### 3.1.6.8 Structure addressing

Pre 5.1 MASM versions used to accept structure member names with no mention of the parent structure, and about any notation, including the '+' sign, etc…
Post-5.1 versions of MASM will only accept to address a structure member through a base register if it has been told that the base register actually points to the relevant structure.

```
Item STRUCT
Foo    DWORD Foo
Bar    DWORD Bar
Item ENDS
```

MyItem  Item <>

One way to do this is by fully qualifying the structure path, such as

```
   MOV EAX,MyItem.Foo
```

Yet another (needlessly clumsy) notation is

```
   MOV EAX,(Item PTR[EBX]).Foo
```

Finally, one can use the ASSUME directive. This way is particularly convenient in the most common case, when several members of the same structure are manipulated in the same code area:

```
   MOV EBX,OFFSET MyItem    ;Get access to MyItem.

   ASSUME EBX:PTR Item               ;Tell MASM what EBX points to.

   MOV EAX,[EBX].Foo         ;Works even if labels Foo and Bar are
   MOV ECX,[EBX].Bar         ;used in many distinct structs.

   ASSUME EBX:Nothing                ;Tell MASM we're done with EBX.
```

This form as the added benefit of checking for erroneous base register use. MASM will flag as an error any attempt to use the wrong register for addressing the structure.

Alternately, it is possible to disable checking through the OPTION OLDSTRUCTS directive, but this is ***not*** recommended: it prevents the use of identical member names in two distinct structures, the nesting of structures and many other useful features. This likely to bite you in some of the numerous Win32 structures, where the identical symbols are often use in distinct structures.

All of this (and some more) is documented in the MASM Programmer's Guide, but the information is once again oddly split in two parts:

- under "Chapter 5/ Structures and Unions/ Referencing Structures, Unions and Fields" Page 126, (where you would expect to find it),
- under "Appendix A, Differences between MASM 6.1 and MASM 5.1, OPTION OLDSTRUCTS (text and examples)", pages 370–371, (where you would probably ***not*** expect to find it…)

### 3.1.6.9 Use of SIZEOF & LENGTHOF

SIZEOF and LENGTHOF replace the old brain-damaged (and hardly usable) SIZE and LENGTH of the previous versions of MASM.
The SIZEOF operator is probably the most useful, and works with about anything the assembler knows of: BYTE strings, arrays, structures, TYPEDEFs, etc…
For instance, SIZEOF DWORD is a valid expression that yields "4."
It is considerably more useful than the previous SIZE operators, and simpler than the "classical" way of computing the length of an item:

```
Foo             BYTE 'Some string here'
FooLng = $-Foo
```

For details about SIZEOF, LENGTHOF and TYPE, see page 108 of the MASM Programmer's Guide.

### 3.1.6.10 Use of TYPEDEF

MASM 6.1x offers a TYPEDEF feature, comparable to the one in C. I ended up deciding to generally not use it, as it introduced a bit too much constraints to my own taste for an assembly language: One has to draw a line between high level and a low level, and my personal decision was to draw it here.
I didn't investigate too much either into the whereabouts of using TYPEDEFs. There is an exception to the rule, though, and this is the use of INVOKE through pointers: This requires the use of TYPEDEF to generate the right calling sequences, see "Chapter 7/ Procedures/ Calling Procedures with Invoke/ Invoking Procedures Indirectly" on top of page 198 in the MASM Programmer's Guide.

### 3.1.6.11 Use of ALIAS

See below, "Weak Externals", page 43.

## 3.1.7 MASM bugs and shortcomings

### 3.1.7.1 Invalid code generation in INVOKE using 16 bit parameters (or a mix of 16 and 32 bit)

***Note: This 6.11d bug has not yet been checked in MASM 6.12***

When one declares an 8 or 16 bit parameter in an INVOKE list, MASM gets very confused: it tries to be smart and to generate code extending the parameter on the stack to 32 bits, but gets hopelessly confused in the ***data size*** of the set of PUSHes and POPs that it generates. The exact error depends on the exact code pattern being assembled, but the net result is always inconsistent generated code, and a stack structure that doesn't match the instructions MASM generated (carefully look at the extended listing).

The result is a GPF that seems to strike from nowhere. This occurs when the current code segment is a 32-bit one ("USE32"), which is always the case when programming for Win32.

The resulting GPF is such that when it happens, it looks like it can't be tracked: ESP gets loaded with an invalid value so when the process crashes, you have completely lost the stack context and/or the value of EIP, and don't know anymore where the error came from. Even single stepping in the code is confusing, as when one experiences this for the first time, the problem seems to strike from nowhere (like a fault that would happen in some unrelated and unknown code portion).

A similar incorrect set of instructions is generated if you mention segment registers in an INVOKE list (not that usual, though).

You have been warned!

### 3.1.7.2 The infamous 512 bytes buffer

There is a ***very annoying*** limitation in ML (MASM): Its input (parsing) buffer, aka "logical line" is only 512 bytes. This is documented as such in the Programmer's Guide, "Chapter 1/ Language Components of MASM/ Statements", page 22, so at this point, we have to call it a feature.

But when coding a Win32 INVOKE with a long parm list, you might want to code (and document your code) such as this:

```
 INVOKE CreateProcess,
        OFFSET lpApplicationName,    ;=> to EXE name
        OFFSET lpCommandLine,        ;=> to command line string
        OFFSET lpProcessAttributes,  ;=> to process sec attribs
        OFFSET lpThreadAttributes,   ;=> to thread sec attribs
        bInheritHandles,             ;handle inheritance flag
        dwCreationFlags,             ;creation flags
        OFFSET lpEnvironment,        ;=> to new env block
        OFFSET lpCurrentDirectory,   ;=> to current dir name
        OFFSET lpStartupInfo,        ;=> to STARTUPINFO
        OFFSET lpProcessInformation  ;=> to PROCESS_INFORMATION
```

Well, forget it; the comments count in the 512 bytes, so the above whole "logical line" (that doesn't use TAB characters but spaces) doesn't fit in the 512 buffer. MASM will flag it as an error (Booo). However long the INVOKE is (and some of them ARE very long), it has to fit in 512 bytes… So you have to remove comments and leading spaces, pack several parameters per line, etc…

You can ***NOT*** get away by using the continuation character (\) nor any other trick.
A similar problem is likely to happen with macros generating long byte strings.
***VERY*** frustrating.

### 3.1.7.3 INVOKE and forward references

INVOKE doesn't take forward references to PROTO definitions, although MASM is defined by Microsoft as an N-pass assembler, and although a CALL instruction does accept forward references.
So you have to move PROTO definitions ahead of their first invocation.
Likewise, ADDR is sometimes required instead of OFFSET in INVOKE parameter lists, but ADDR does not support forward references (while OFFSET does).

### 3.1.7.4 Macro limitations

MASM doesn't recognize strings and complains about unmatched items when it finds symbols such as '<' and '(' in string parameters.

### 3.1.7.5 Listing generation

A number of minor syntax errors generate a "fatal error." An example is typing something like

```
Foo PROC USES EAX,ESI           ;Extraneous ","
```

but there are many other potential (and very benign) causes for a fatal error.
Unfortunately, the Fatal Error prevents the .LST (listing) file from being generated. This is particularly painful when debugging complex macros, since the only way to debug macros is precisely to generate a listing with full code expansion.
So one ends up with a macro that generate offending code that can't be seen because the generated code can't be listed. Yet another ***very*** frustrating situation.

### 3.1.7.6 Missing conditions in structuring directives

This one belongs to the "shortcomings" category.
As documented in the MASM Programmer's Manual, "Chapter 7/ Loops/ Writing Loop Conditions/ Expression Operators", page 178, all possible conditions can be tested by creating complex expressions, such as in

```
  .IF (EAX > 0) && (DWORD PTR [EBX] == 0)
  NOP
  .ENDIF
```

that generates

```
 00000052   2   83 F8 00  *       cmp    eax, 000h
 00000055 7m,3  76 06     *       jbe    @C0006
 00000057   5   83 3B 00  *       cmp    dword ptr [ebx], 000h
 0000005A 7m,3  75 01     *       jne    @C0006
 0000005C   3   90                nop
                          .ENDIF
 0000005D              *@C0006:
```

But quite often, one needs to test preexisting condition codes, such as those resulting from an arithmetic operation (e.g. SUB EAX,EBX), or those returned by a routine.
To handle these cases, the authors of MASM created some special (and somewhat redundant) symbolic for ***directly*** testing preexisting condition flags: ZERO?, CARRY?, OVERFLOW?, SIGN? and PARITY?.

They can be used as in

```
.IF CARRY?              ;Generates a JNC
```

The MASM authors apparently didn't think about the obvious, that of ***simply deriving all the existing*** Intel J<cond> mnemonics for simple condition testing in their structured programming directive: allowing expressions such as .IF Z?, .WHILE C?, .UNTIL S?, .BREAK .IF P?, CONTINUE IF AO?, etc,... would have been simple, intuitive and exhaustive.

This oversight is unfortunate for two reasons:

First,

```
.IF !CARRY?          ; The only direct way to generate a JC
```

is not as readable as

```
.IF ABOVE?           ; Is more mnemonic.
```

At the opposite of MASM, the Intel mnemonics define various synonyms for the same conditions to improve code readability. For instance, Intel defines both a “JZ” (Zero) and a “JE” (Equal), that are exactly the same instruction. But testing for ZERO makes sense after a subtraction while testing for EQUAL is intuitive after a comparison. Ditto for JL and JNGE, JGE and JNL, etc…

But the most annoying part is this story is that the existing predefined symbols don’t allow generation of some of the less usual “combo” jumps, those that test for multiple conditions at once, such as G, GE, BE, LE, and their negations. I have not found any way to solve this one.
Even trying to use combined flags does not work: a “JL” for instance takes a jump when the SIGN? flag is not equal to the OVERFLOW? flag.
Let’s try this:

```
.IF Sign? == Overflow?      ;This should generate a "JL"
```

Ooops!

```
error A2154: syntax error in control-flow directive
 00000060 7m,3  75 01          *      jne    @C0008
```

(Boooo!)…

Another example: A “JA” takes its jump if both Carry and Zero are false. The inverse of a JA is a JBE, and is taken when either Carry or Zero is True.
So the following expression should generate a “JA”:

```
   .IF CARRY? || ZERO?  ;Should generate a JA
```

But instead, it generates the right logical sequence in the most inefficient way.

```
 00000060 7m,3  72 02  *  jb     @C0009
 00000062 7m,3  75 01  *  jne    @C0008
```

The bottom line is that we have no way to generate any of the jump instructions that test combined flags, nor to redefine the right mnemonics for them.

## 3.1.7.7 Major flaws in the MASM macro language

The macro language as it is in MASM 6.1x is barely usable for complex macros that need to perform true parsing tasks to generate complex structures or streams. A good macro language should be able to simply describe the syntax of any instruction or directive consistent with the existing language. This is far from being the case with MASM.

The successive piling of features since the original MASM 1.0 (back in 1981!) resulted in what the macro language is today, that is a rather inconsistent, limited and much too complex language.
The current macro language does not allow the creation of macro instructions that would behave exactly as native instructions, directive and/or data definition streams: the macro notation comes in the way.
Literals passed as operands to macro instruction that would take a list of operands the same way as a BYTE directive, for instance, can’t include special symbols such as “!”, “<” and “>”, because they have special meanings to the macro language. The “!” is the forcing character for parameters of macros, while the rule is different in the rest of MASM: For instance, forcing a quote in a quoted string is forced by doubling the quote, as in :

```
   BYTE “Foo “”Bar””  “
```

The use of some older features of the macro language additionally preclude the use of the quote, double quote, backslash, percent and ampersand symbols.
The macro syntax does not allow one to parse the “label” field of a macro invocation as a parameter and use it as such to generate a label somewhere in the generated code. Etc…

As an example, here is a problem to solve:
Use the macro language to define a directive that would transparently generate a Unicode strings using a syntax exactly compatible with that of the native BYTE directive, for instance.
Go to “The String macro”, page 51, for the best solution we found so far. And remember that the result does not reflect in any way the pain it was to achieve it. The

challenge is open, by the way: anyone with a better solution to the problem, please Email!

The way things are, I don't see any way this situation could be fixed by "enhancing" the macro language again.
If MASM ever goes back to the development cycle one day, or if a new MASM compatible assembler is ever developed, I would gladly vote for the creation of a brand new, ***incompatible*** but consistent macro language, and rewrite all my existing macros from scratch so they can still compile the old source code syntax. I guess such an enhanced MASM could also insure backward compatibility by taking an OPTION OLDMACROS directive to hide the new syntax and re-enable the existing brain-damaged 6.1x syntax.

## 3.2 Using LINK

### 3.2.1 Libraries

Use the /LIBPATH: switch to specify a single directory path where default libraries can be found. If you need to specify multiple directories, you need to use the /LIBPATH: switch several times.
The default library files that will be looked for using these path will typically be those you specified using INCLUDELIB statements in your prototype include files (see 3.1.3.3, "Win32 (and other) function prototypes".
This is the simplest way to let the linker reach the Win32 import libraries from the Win32SDK:

/LIBPATH:C:\Win32SDK\LIB

### 3.2.2 Debugging options

For debugging, use the three following switches:
/DEBUG
/DEBUGTYPE:CV
/PDB:none.

Don't use
DEBUGTYPE:COFF or
DEBUGTYPE:BOTH.

The "real" full-featured debugging information that symbolic debuggers use is actually CodeView format information, so /DEBUGTYPE:CV should be all you need.
In addition, using the COFF debugging information will sometimes (depending on some obscure pattern in you code) result in a failed ASSERT in LINK.EXE (crash), apparently due to some disagreement on the COFF debug records between ML and LINK (***Note: The 6.12 README.TXT mentions that work has been done to fix problems in that area, but we have not checked it yet***).
Using PDB:NONE includes the debugging info into the .EXE file, which is probably the simplest way to have it: all the debugging information the debugger needs is in the

.EXE file, the debugger doesn't have to locate and open a separate /PDB file. Using a separate PDB file is fine, and will make the .EXE much smaller, but adding the risk of having de synchronized .EXE and debugging info, and leaving it to you to make sure the debugger finds the .PDB file.

### 3.2.3 Linking an .EXE file

#### 3.2.3.1 Linking a Console executable

Use the /SUBSYSTEM:CONSOLE switch.

The (large) capabilities accessible to Console programs are defined in the Win32 SDK (MSDN Level 2 and upper), in

Win32 Programmer's Reference
  Overviews,
    System Services
      Consoles and Character-Mode Support

#### 3.2.3.2 Linking a Windows executable

Use the /SUBSYSTEM:WINDOWS switch.

### 3.2.4 Linking a DLL file

Use the /DLL switch.
You might also add the /SUBSYSTEM:WINDOWS or /SUBSYSTEM:CONSOLE switches. If you don't do it, according to the generated PE code, Link will assume /SUBSYSTEM:WINDOWS.
The subsystem switch definitely reflects in the generated PE file, as can be told by running a PE dump program. But to the best of my knowledge, this does not makes any difference to the Win32 loaders, though: Under Win95, a console application doesn't care to call functions in a DLL declared as Windows. I have not checked at this point whether it makes any difference to a Windows application that the DLL is declared as Console. Nor did I test any of these under NT.
But I don't see any reason why it would matter, as many Windows DLLs are used by both console and windows programs, and I have never heard about any special PE restriction.

About everything you need to know about DLLs is defined in the Win32 SDK CD-ROM (MSDN level 2 and upper). It's more precisely defined in:

Win32 Programmer's Reference
  Overviews
    System Services
      Dynamic-Link Library

The above section explains how a DLL is defined by an entry point function (with its initialization / exit sub-functions), and a number of exported functions the DLL exposes to the outside world.

Another great place for DLL information is [Richter 97.01]. It specially covers what you ***should not, ever*** do in one of the DLLEntryPoint routines, and why not knowing it can nicely deadlock you program. I am not sure this later fact is documented anywhere else.

The only other thing you need to realize to connect the above pieces is that the name of the entry point function is defined through the /ENTRY: directive of the LINK utility (the same directive that is used to define the program/process entry point in an .EXE).

At this point, you will have about all the first level information you need to know about DLLs, and especially about the entry point function.

The rest of what you need explains how to tell LINK to generate your DLL, and this is documented in the MSDN Library CD-ROM:

Product Documentation
  Languages
    Visual C++ x.y
      User's Guides
        Visual C++ User's Guide
          LINK Reference
            Module-Definition (.DEF) Files

The functions your DLL expose will be defined in the EXPORTS section of the .DEF files. Alternately, the other way to define exports is through the use of a command line switches (/EXPORT:). For any large project, I tend to like the .DEF file approach better, but this is largely a matter of personal preference (and of the building tools one uses).

As you can see, we picked up the documentation of the VC++ linker that we took from recent MSDN Library documentation. About the same documentation applies to several earlier versions of the 32-bit LINK utility, probably up to its earliest release that used to be known as LINK32.

### 3.2.5 Advanced linking techniques

#### 3.2.5.1 Grouped Sections

There is a little know characteristic in the COFF/PE specifications (MSDN, “Portable Executable and Common Object File Format (PE/COFF) Specification 4.1”) that can prove very useful for program construction.
Here is an excerpt of the specification:

The “$” character (dollar sign) has a special interpretation in section names in object files.

When determining the image section that will contain the contents of an object section, the linker discards the "$" and all characters following it. Thus, an object section named **.text$X** will actually contribute to the **.text** section in the image.
However, the characters following the "$" determine the ordering of the contributions to the image section. All contributions with the same object-section name will be allocated contiguously in the image, and the blocks of contributions will be sorted in lexical order by object-section name. Therefore, everything in object files with section name **.text$X** will end up together, after the **.text$W** contributions and before the **.text$Y** contributions.
The section name in an image file will never contain a "$" character.

Using this feature, the linker can be used to construct tables of related objects. The related objects can be declared in different modules, but the linker will be able to consolidate / concatenate them in an orderly way in the resulting image, building structures that the program will be able to use at run time.

An example of this use can be found in ": Runtime Initialization / Termination Macros", page 56

## 3.2.5.2 DLL forwarders

First, the good news:

The PE format offers a very interesting (but barely documented) option to DLL creators: DLL forwarders. A DLL forwarder allows the DLL programmer to specify an entry point in a DLL and "forward" the DLL call at runtime to another function in another DLL. A typical use for this technique is in implementing a "shell" to another DLL, for instance, that would implement front end to certain functions and transparently forward the call of others to existing functions in the shelled DLL. The nice thing about function forwarders is that being implemented in the image format and in the loader, they don't require the writing of a single line of code. In addition, the OS overhead of function forwarding is minimal.

The only place this capability is documented in the official Microsoft documentation is in the MSDN library, under

Specifications
  Portable Executable and Common Object File Format (PE/COFF) Specification 4.1

Unfortunately, while the PE/COFF documentation describes how this capability is implemented in the PE file, it doesn't describe how to talk any given linker into generating the corresponding image feature. And of course, the official MS Link documentation doesn't describe how to instruct LINK to create a DLL forwarder for a given DLL entry.
Matt Pietrek mentioned the existence of this feature in [Pietrek 95.01] as well as in several articles in the Microsoft Systems Journal, but never got around to study how the feature could be implemented using Microsoft Link.
Jeffrey Richter finally provided the answer in [Richter 96.01].

From a .DEF file, the general definition of an EXPORTS table entry looks like:

*entryname*[=*internalname*] [@*ordinal*[**NONAME**]] [**DATA**] [**PRIVATE**]

The trick is a very simple one: A DLL forwarder is created by placing the address of the target function as the optional ***internalname***, such as:

```
SomeFunc=OtherDLL.SomeOtherFunc
```

When defining exports from the command line, the syntax is:

```
/export:SomeFunc=OtherDLL.SomeOtherFunc
```

I have found that at least with my MASM / linker couple, using this feature forced me to specify decorated names in the .DEF file where the forwarder was defined. This might or might not be true of other versions of the software. Without using forwarders, you don't have to bother as the linker handles the decoration automatically.

If you get undefined symbols corresponding to names involved in forwarders, try decorating the names manually. This will likely propagate the error up the DLL chain, back to the main .EXE, and you will have to fix the upstream .DEF files accordingly.

Now for the bad news:

DLL forwarders are ***not*** implemented in the Win95 loader. You will get an OS-generated runtime error complaining about a missing DLL symbol if you try to use them.

### 3.2.5.3 Weak Externals

Weak Externals are a way to provide link-time default replacements to undefined externals.

Excerpt from MSDN, "Portable Executable and Common Object File Format (PE/COFF) Specification 4.1"

> "Weak externals" are a mechanism for object files allowing flexibility at link time. A module can contain an unresolved external symbol (sym1), but it can also include an auxiliary record indicating that if sym1 is not present at link time, another external symbol (sym2) is used to resolve references instead.
>
> If a definition of sym1 is linked, then an external reference to the symbol is resolved normally. If a definition of sym1 is not linked, then all references to the weak external for sym1 refer to sym2 instead. The external symbol, sym2, must always be linked; typically it is defined in the module containing the weak reference to sym1.

Weak external can be implemented in MASM 6.1x using the ALIAS directive (only very poorly described in a MASM 6.11 release note). In the example described above, the directive would be:

```
ALIAS <sym2> = <sym1>
```

Beware: Any syntax errors in a ALIAS definitions (and/or reference to a missing symbol) usually trigger page faults in MASM 6.11d (Owell…).

***Note: The README.TXT file for MASM 6.12 claims that a number of Access Violation causes have been fixed, but we have not checked this one at this time.***

## 3.3 Debugging an assembly language Win32 application

If you followed the above rules and used the options mentioned above for assembling and linking, you ended up with an executable with full CodeView debug information.

Any debugger supporting CodeView debugging info should be able to support your executable file and offer full symbolic / source debugging. I have successfully used the WATCOM debugger (version 10.6), Microsoft's Developer Studio debugger and Numega's SoftIce 3.x.

If Visual Studio (aka Developer Studio) is loaded on your machine, you can use it as a mere debugger, even if you never use its IDE to develop your MASM application: If you want to debug your great new FUBAR.EXE application on the current drive, and providing you installed VC++ on drive G: under \MSDEV, just run:

```
G:\MSDEV\BIN\MSDEV.EXE FUBAR.EXE
```

This will launch the Visual Studio IDE right into the debugger and allow you to start debugging. All symbolic facilities should be there.

The best tool I have found so far for debugging Win32 applications is Numega's SoftIce 3.x., and the best setup I found for it was to run it on a single machine, using a second video controller and dedicating a small alternate screen to the debugger.

Unfortunately, although it claims to fully support MASM, SoftICE doesn't support the complete legal character set that MASM allows (as defined in MASM Programmer's Guide, "Chapter 1/ Language Components of MASM/ Identifiers" page 9): as a result, labels including special characters such as '$' and '?' are not supported and can not be accessed or used in expressions with SoftICE. This is quite unfortunate, since MASM itself ***does*** generates labels with '?' symbols for local symbols – such as those generated by the structured programming macros.
In addition, there are quite a few libraries that use the perfectly legal '$' and '?' characters, and debugging code using these libraries is very akward.

# 4 Various gripes

## 4.1 The absence of LDT support in Intel-based platforms

Microsoft decided a few years ago that since NT was to be multi-platform, the only MS blessed way of programming was to use C (or C++). From that day, Microsoft seemingly stopped caring about assembly language, to the point of mostly ignoring it as they do today. What MS might not have anticipated is that the MIPS people, soon followed by the PowerPC folks, would drop off the NT market, and that the remaining non-Intel NT platform (the DEC Alpha) would represent a minuscule part of the market, making the whole portability issue a very moot point.

The MS folks went as far as preventing use of a key feature of the Intel CPU, seemingly because they didn't have any exact counterpart on other (RISC) platforms.

Did you ever notice that there is no way to benefit from segmentation in user mode under WIntel32 platforms?
I know that the forced use of segmentation with 16-bitness in previous times gave a terrible reputation to segmentation: It was a nightmare in 16BitLand to manipulate large pieces of data broken in 64k segments.

But segments have another use and benefit:
They provide a very efficient way to implement multi-instantiation: By simply changing the place where the data segment registers point it is possible to implement code reentrancy and re-instantiation in a fully code-transparent transparent way. And as the segment registers are part of the CPU context, the OS can automatically keep separate data context for each thread running off the same piece of object code.
So by initially manipulating the segment registers, a process thread could launch many threads running the same code, with each thread running off its own data area (materialized by its own descriptor) The benefit is, no addressing restriction nor inefficiency in the available addressing modes. This segment mechanism was explictely designed in the CPU as the simplest and one of the most efficient way to program a piece of code running a single application for many users simultaneously, for instance.

Well, the problem is that Win32 provides no documented way to let a Ring 3 (user mode) process allocate an LDT entry. This means one can't allocate a bunch of memory from the OS, ask the OS to create a selector for it and start a new instance of a thread that would use this new data area as its data segment.

The only mechanism that Microsoft offers to achieve multi-instantiation is what they call Thread Local Storage (TLS): It is implemented in two different ways, Dynamic TLS and Static TLS (see [Richter 97.01] for details).
Dynamic TLS allows the programmer to use an OS allocated 64 DWORD array to maintain thread-specific pointer. This implies:

- That the number of data items that can be tracked this way is limited to 64,
- that all instantiated memory accesses are done at best through an indirection
- that access to the array is accomplished by system calls, making the mechanism even less efficient

The least inefficient way, static TLS, is still hardly acceptable: Compile-time storage is allocated in the .TLS section, and the OS replicate the TLS segment for each new thread that is started. This means that ***each*** thread in the process gets a block of TLS storage the size of ***all*** the TLS data from all the threads… In other words, the TLS section is allocated as if it where global ***for all the threads***, and the threads that don't need any instantiation data still get it.
In addition, and as pointed out by [Richter 97.01], "*on an x86 CPU, three additional machine instructions are generated for* ***every*** *reference to a static TLS variable.*"
This is something most C programmers don't see, since the extra overhead only appears in the code generated by the compiler, that most C programmers don't look at or really care about. ***The problem is completely hidden at the C source level***. But it certainly doesn't look so to the assembly language programmer, and the efficiency of the resulting code is obviously much lower. Furthermore, using TLS, the programmer looses in the process the automatic inter-thread protection the use of segmentation

would have provided: attempts to access data outside a memory segment is something an Intel CPU automatically trap.
Finally, static TLS can only be used with implicitly loaded DLLs. No Win32 OS is able to properly initialize TLS storage for explicitly loaded DLLs (loaded via LoadLibrary). For details, see "Static Thread-Local Storage", [Richter 97.01].

The bottom line is that, at least for the Intel implementation, TLS is hardly more than a dirty and ***very*** inefficient kludge.

Apart from TLS, the only other official way to achieve multi-instantiation in theWin32 world is to create multiple processes (rather than multiple threads). This belongs to the steam-hammer action category though, and doesn't compete for efficiency with the lightweight multi-threaded way:

- Each process requires a separate .EXE file,
- Each process requires reloading / remapping the same memory image for each instantiation of the process.
- Changing process context is more costly than changing thread context (all process-local items are part of the process context and don't need to be changed when switching thread context inside the same process).
- Using separate processes precludes the use of any of the lightweight intra-process synchronization and data sharing mechanisms such as global memory items, critical sections, etc,… And since the inter-process synchronization / communication mechanisms have to cross process boundaries, they are much more costly than the intra-process ones.

Whenever we had a chance to ask, we only got two explanations so far from Microsoft personnel about this missing feature:

The first one is the "flat model" dogma: "Win32 uses the flat model, and this model precludes the use of segmentation."
This is simply not true. Using the flat model never prevented the use of segmentation, as the Intel CPU documentation clearly states. There is no such thing in the Intel CPU as a "flat model bit", and using a flat model is a pure programming convention / convenience. No CPU-inherent technical limitation prevents a programmer from occasionally using a segment register for any reason. As we mentioned above, the best evidence is that Microsoft themselves use segmentation in the Win32 world: in any Win32 thread, the FS register always contains a special descriptor, that doesn't follow the flat model rules, and is used to access the TID (Thread Information Block, see [Pietrek 95.01]). ***The lack of access to segmentation from Ring 3 code only comes from an OS design decision***.

The second explanation Microsoft commonly gives about the lack of access to segmentation from Ring 3 code is the need for portability, and the lack of hardware mechanisms to implement segmentation on non-Intel platforms. As we already mentioned, this looks to us as a very moot point, as

- portability is non-existent in the Win9x world anyway, since there is no such thing as a non-Intel-compatible Win9x platform, and conversely Win9x-specific features are only supported on Wintel32 platforms,

- portability is of little use in the NT world, and it is even diminishing: cumulated sales of NT on non-Intel platforms are a milli- or micro-market, even shrinking now the MIPS and the PowerPC contenders threw the towel,
- there is only one non-Intel machine left in the arena (the Digital Alpha), it is currently marginal, and this situation doesn't seem likely to improve: AutoDesk, makers of AutoCAD, one of the major selling application for the Alpha, recently decided to drop support for the NT/Alpha platform: there was not enough demand…
- from an API point of view, it would only take two or three Intel-specific NT system calls to implement this mechanism, and last but not least,
- it's ultimately the responsibility of the people designing application software to decide whether portability is (or is not) a more desirable goal to achieve than efficiency on any given platform they decide to choose.

We think there is clearly a case here for Microsoft implementing a few Intel-specific syscalls and allowing proper thread instantiation through the use of selectors and Ring 3 LDT manipulation.

Since we are not likely to see Microsoft change their position on this point in the short term (*!*), here are some alternate solutions for those assembly language programmers that the TLS kludges do not satisfy:

One way could be to restrict all data to local (stack based) storage, but there are some problems with that approach:

- it severely limits the addressing modes available to the programmer. This is particularly limiting when complex, nested structures / arrays need to be accessed,
- it imposes severe architectural constraint to the programmer: there are many programming situations where global data access is required, specially in time-critical real-time applications, when the database is large enough. A number of Win32 constructs actually ***require*** global data access,

Another variant could be:

1. Program the thread to instantiate, grouping its instance data together,
2. compute at runtime the size of the static RAM database for the thread,
3. allocate a chunk of memory of the same size,
4. initialize the new chunk as needed (possibly by a mere memory move from the original to the new chunk),
5. compute the offset between the original memory block and the new chunk,
6. load a base register with the result and
7. address each and every instantiated variable through based addressing, using a different chunk (with a different base) for each thread to instantiate.

This solution is slightly better: it provides a “global” database with no size limitation, while still leaving stack based addressing to parameter passing and true local storage. But still far from ideal:

- it sacrifices a precious and scarce base register for the whole life of the thread,
- it precludes any access to the direct addressing mode, the simplest an least error prone of all,
- it prevents the use of base + index addressing inside the thread instantiated data block (since base is already used to maintain access to the data block), seriously complicating access to complex data structures, and
- if the programmer forgets to use base addressing on **any single** instruction, the program will access the RAM instance of the ***original*** thread rather than that of the thread it’s running in (ouch), creating very hard to track bugs. But everything being relative, keep in mind that the same kind of error is even more likely to happen using the much more twisted TLS addressing ways.

The irony is that nearly the same logic as we described above would apply if LDT selector allocation were allowed: step 5 above (and following ones) would be replaced by something like

5. Allocate an LDT selector
6. Load a segment register with the result
7. Address each and every variable just as you would if this thread were alone: It ***is*** actually alone to access this memory segment.

The main difference is that each memory access could be then be achieved safely and efficiently using any and all of the addressing methods the CPU offers.

# 5 Win32ASM Toolkit

The toolkit contains an undefined number of files. Undefined, because

- the toolkit is a never ending work,
- the documentation always tend to lag behind,
- We have already postponed the release of this document too much, waiting for planned enhancements that did not make it,
- it is probably better to provide a file with no documentation (or an obsolete documentation) than no file at all,
- these files should be considered as “work in progress.”

For these reasons, we can not guarantee that files associated with this documentation match exactly what is described, nor can we insure that they are fully stable. In other words, and as stated in the disclaimer ahead of this document, you are using this code at your own risks…

## 5.1 The Example files

There are at least two example files with source code along with this document:

Win32Proto and Win32DLL.

Win32Proto is a skeleton Win32 program including a tool bar, some dialog boxes, and a small bunch of gadgets. It can be used as a framework for real projects.

Win32DLL demonstrates how to build a 100% MASM DLL, and also how a DLL can be called from a 100% MASM .EXE program.

See the corresponding subdirectories and the included README files for more details.

## 5.2 The include files

## 5.2.1 General Include files

### 5.2.1.1 Win32Inc.equ

This include file is generally found ahead of each and every Win32ASM application module.
It includes four other include files, that are needed in about all circumstances:

```
   Include UnicAnsi.EQU     ;Unicode / ANSI stuff
   Include Win32Types.EQU   ;Various typedefs
   Include Win32Defs.EQU    ;Many equates
   Include Win32Strs.EQU    ;Various Windows structures
```

These four files and their contents are described below.

#### 5.2.1.1.1 UnicAnsi.equ

This include file handles the character set issues. At the time I am writing this, I am far from having covered, or even started studying the topic: All my current works has to be Win95 compatible, and I thus I have to stick to ANSI representation (Win95 does not support Unicode format). So the only part of UnicAnsi.equ I am actually using today is the UnicAnsiExtern macro (see below).

Most of the other material in this file directly comes from Sven B. Schreiber’s Walk32 work, mentioned in this document, and has not yet been used or tested in the Win32ASM environment. I could even have broken it while reshuffling it around and not have realized it yet.

##### 5.2.1.1.1.1 The UnicAnsiExtern macro:

Sven resolved the Unicode/ANSI issues at link time, in his own linker. Since I had to use the Microsoft linker, I had to solve the problem another way.
I ended up doing it about the same way Microsoft does it in their “C” headers, by using a compile time macro.

Win32ASM uses the UnicAnsiExtern macro to turn a function name into a TEXTEQU containing its character set dependant name. The macro uses the Unicode switch to postpend an “A” (for ANSI) or a “W” (for Wide) and compose the true external name of the function.
The UnicAnsiExtern macro has to reference each character set dependent function ahead of its PROTO definition. You will find groups of UnicAnsiExtern macros ahead of any Win32 API include file that contains charset dependent functions.

At assembly time, any reference to the generic name of a function is automatically changed by the UnicAnsiExtern macro into the relevant charset dependent name.

If during your own development, you add a Win32 PROTO to some Win32 API equate file and get a undefined reference at link time, double-check the function name in the Win32SDK: if you spelled it properly (case included), you might have hit a function that is character set dependent. In this case, add an UnicAnsiExtern entry with the name of the new function ahead of the equate file, before your PROTO definition.

###### 5.2.1.1.1.2 The String macro

##### 5.2.1.1.2 Win32Types.equ

This file contains various TYPEDEF definitions.
A number of these TYPEDEFs are used in various structures in the Win32Strs.equ file.

##### 5.2.1.1.3 Win32Defs.equ

This file contains miscellaneous Win32 EQUate and TEXTEQU definitions.

##### 5.2.1.1.4 Win32Strs.equ

This file contains a number of Win32 structure definitions.

#### 5.2.1.2 Win32Res.equ

This file contains numerous EQUates related to resource definitions.

### 5.2.2 API header include files

These include correspond to Win32 DLLs and their Import libraries.
Each .equ file contains

An INCLUDELIB referencing the import lib, instructing LINK to pull it in,
the function headers (PROTOs) corresponding to the .DLL of the same name,
related structures, equates and constants.

I only started to organize the API header include files this way recently. In addition, I started by using existing include files for Win equates: First, the file provided by Microsoft in the DDK and derived from the Winbase.h file, then later those compiled by Sven B. Schreiber in Walk32. And finally, a number of equates and structures are not specific to a specific DLL but are used instead by several.
As a result, a number of structures (too many structures), equates and constants are not located in the .equ file they should belong to, but can be found in one of the “General Include files” (see 5.2.1) instead.

In addition, as I explained above, I only create and fill these files on a “as needed” basis. So they can in no way be considered exhaustive. The WinMM.equ file, for instance, contains a single PROTO definition at the time I am writing this. Since my

programming is more oriented toward console (service) applications, not that much of the Windows API is covered at this point.
This situation should slowly improve with time, as I keep on adding new functions, structures and reorganizing this file set as needed.

#### 5.2.2.1 CommCtl32.equ

#### 5.2.2.2 CommDlg32.equ

#### 5.2.2.3 GDI32.equ

#### 5.2.2.4 Kernel32.equ

#### 5.2.2.5 TAPI32.equ

#### 5.2.2.6 User32.equ

#### 5.2.2.7 WinMM.equ

#### 5.2.2.8 WinSpool.equ

## 5.3 The macro files

### 5.3.1 Instr.mac

INSTR.MAC contains various utility macros. Some of them more or less extend the set of instruction / directives of MASM, thus the name of the macro file.

#### 5.3.1.1 Structuring directive extensions

##### 5.3.1.1.1 .BLOCK & ENDBLOCK

.BLOCK and .ENDBLOCK generate no code. They simply define a block of code with a single exit point, located after the ENDBLOCK. One can jump out of a BLOCK through the regular structuring directives, such as .BREAK, .BREAK .IF, etc…
.BLOCK is a synonym for ".REPEAT" and .ENDBLOCK a synonym for ".UNTIL 1," but the .BLOCK and .ENDBLOCK name make their purpose more obvious and increase code readability.

```
   .BLOCK                  ;We just sent DLE ++ DLE 0.
   CALL FBIIGetParms  ;get other end's parms,
   .BREAK .IF CARRY? ;Drop out  if error.
   CALL FBIISendParms            ;queue our parameters packet,
   .BREAK .IF CARRY?
   CALL FBIxWait4Tx    ;Wait until we're acked to go to data
   .BREAK .IF CARRY?
   CALL FBIxRelTxSem ;Release the Tx semaphore we just used.
   CALL FBTxQuotePatch          ;Update the quote table.
   CALL FBTxInit1Init    ;Initiator, end of init, Rxer and Txer.
   CALL FBRxInit1Init
   CLC
   .ENDBLOCK
```

## 5.3.1.1.2 FOREVER

.FOREVER is a termination for a .REPEAT loop that unconditionally jumps back to the head of the loop. It is synonym to “.UNTIL 0”, but here again; readability is the key:
.REPEAT / .FOREVER is simply more explicit than .REPEAT / UNTIL 0.

```
   .REPEAT
   INVOKE GetMessage,
          OFFSET winMsg,    ;Adress of Msg structure,
          0,              ;Window to get msg from,
          0,              ;Filter min,
          0               ;filter max.
   .BREAK .IF (EAX == 0)      ;Can this ever happen here?...
   INC StatTAPIMsgs          ;Count messages we see.
;  $Display 'Got a Win message',$EOL
   INVOKE DispatchMessage,    ;Dispatch msg to proper winproc, and
          OFFSET winMsg     ;loop again.
   .FOREVER
```

## 5.3.1.1.3 Condition mnemonics in structuring directives

We have seen above in “Missing conditions in structuring directives”, page 36, that the mnemonics the structuring directives allow are very limited set of condition mnemonics and that the structuring directives don’t allow the direct generation of all legal jumps the Intel CPUs can handle.
We can’t fix the latter, but we can slightly improve the former: The INSTR32.MAC file adds 3 mnemonics:

| | |
|---|---|
| EQUAL? | Synonym of ZERO? |
| BELOW? | Synonym of  CARRY? |
| ABOVEorEQUAL | Synonym of !CARRY? |

## 5.3.1.2 Saving and restoring registers

The SAVE and RESTORE macros generate multiple PUSH and POP instructions. They are designed in such a way that the same list of registers (in the same order) can be used for both SAVE and RESTORE, reducing a possible cause for errors.
In addition, the “F” register is handled specially and generates the right instruction (PUSHFD or POPFD).

```
        SAVE EAX,EBX,EDI
        CALL FooBar
        RESTORE EAX,EBX,EDI
```

### 5.3.1.3 UnusedParm

The UnusedParm macro is useful in PROCs where some entry parameters are defined but not used. This happens very often in CALLBACK procedures, for instance.
In this case, if the warning level is set to the maximum value as it should, MASM generates a warning.
UnusedParm allows you to disable the warning (and document the fact that actually not using the parameter is not a bug).

```
   MSGPROC WinProcCMD_ID_HELP_ABOUT

   INVOKE DialogBoxParam,
        hInst,            ;Process instance,
        IDD_ABOUTBOX,     ;"About" box template resource,
        hWnd,             ;owner window,
        OFFSET AboutDlgProc, ;dialog box procedure,
        0                 ;lparam for WM_DIALOGBOX message.
   XOR EAX,EAX
   RET

   UnusedParm wMsg
   UnusedParm wParam
   UnusedParm lParam

WinProcCMD_ID_HELP_ABOUT ENDP
```

### 5.3.1.4 Internal consistency checking macros

These macros are more or less equivalent to the ASSERT macros present in some HLL. The only reason their name is not ASSERT is that I created them before ASSERT became popular in C (and yes, we’re still talking about a post-neolithic era).

All of these macros take a condition code as their first parameter. As the name of the macro imply, the condition mentioned must be realized, or something terrible will happen. For the MUSTBE family of macros, failure will yield a call to a FatalError routine. As the name suggests, FatalError ends up killing the calling process. But before doing so, it displays as much pertinent information about the problem as it can, the minimum being the address where the problem occured. The minimum FatalError routine is a breakpoint (INT 3), that will either directly invoke a debugger (if one capable enough is present in the machine) or invoke the OS “process abend” routine, that will display the registers at the time of the problem (and optionally offer to invoke a debugger). The top of stack will show the address where the INT 3 occured.
There is a more powerful, full fledged FatalError routine as part of this package (see page 60).

The other variations of the MUSTBE macro take a second parameter, a message that is displayed before the faulty process exits.

### 5.3.1.4.1 MUSTBE

This routine does not take any message. The only data the FatalError routine has are the contents of the registers and condition codes, and the address where the problem occurred (sitting at the top of the stack).

```
AND EAX,OnLine       ;Already online?
MUSTBE Z             ;Yes, can't be here. Just crash.
```

### 5.3.1.4.2 MUSTBEM

This routine takes an additional, optional parameter, an error message string. The message is generated in the .CONST section, headed by a byte length, and an INVOKE FatalError is generated, with a pointer to the aforementionned message.

```
CMP hCall,0          ;Check call handle:
MUSTBEM E,'FoolineMakeCall: Call handle already active ?!'
```

### 5.3.1.4.3 MUSTBEMGLE

Same as MUSTBEM, but the macro invokes a variation of the FatalError routine, FatalErrorGLE. FatalErrorGLE invokes GetLastError, formats the corresponding OS error message and present that information together with all the relevant information available to the regular FatalError routine.

```
INVOKE SetConsoleCtrlHandler,
        ADDR BruteForceExit,
        TRUE
OR EAX,EAX
MUSTBEMGLE NZ,'FooMain: SetConsoleControlHandler failed'
```

### 5.3.1.4.4 SHOULDBE

This macro is in essence equivalent to the MUSTBE macro with a major difference: it invokes a “Warning” routine instead of a “FatalError” routine, and “Warning” is supposed to return to the point it was called without changing any register or condition code. Although I have had the “SHOULDBE” macro around forever, I never got around implementing the Warning routine. The FatalError ended up being sufficient for my own use.

### 5.3.1.5 Enumeration macros

The enumeration macro allows the generation of a 0 based sequence of numeric equates, with associated symbols. This macro is useful for defining symbolic corresponding to offset of entries into tables, arrays, etc…

ENUM defines the beginning of an enumeration. It takes a type of data as its required parameter. This can be a predefined type (such as WORD or DWORD), a structure, a typdef, etc…
ENUMITEM takes a symbolic name as its required parameter. It will assign the name to the generated offset.

ENUMEND defines the end of the enumeration table.

Example of use:

```
ENUM WORD
  ENUMITEM FBTxStIdle   ;TXer does plenty of nothing.
  ENUMITEM FBTxStHead2  ;About to send 'B', second header byte.
  ENUMITEM FBTxStData   ;About to send a data byte.
  ENUMITEM FBTxStCRC8   ;TxEr about to send CRC-8 (negociation).
  ENUMITEM FBTxStCRC16L ;TxEr about to send CRC-16, LSB
  ENUMITEM FBTxStCRC16M ;TxEr about to send CRC-16, MSB
  ENUMITEM FBTxStDLEd   ;TXer about to send DLE'd data.
  ENUMITEM FBTxStRing   ;Transmitting from ring buffer.
  ENUMITEM FBTxStChain  ;Txer about to check for chained
                        ;transmission.
ENUMEND
```

The above example will equate FBTxStIdle to 0, FBTxStHead2 to 2, etc,…

### 5.3.1.6 Breakpoint macros

Debug generates an INT 1 (debugger call). Executing it normally results in the debugger getting the focus.

Break is the regular INT 3 breakpoint.
Break2 generates a software NMI, INT 2.

If your machine contains a debugger configured to trap either breakpoint, executing Break or Break2 will wake up the debugger.
Otherwise, the OS will raise the exception dialog box (which might give you a chance to raise the debugger).

## 5.3.2 InitExit.mac: Runtime Initialization / Termination Macros

These macros are designed to resolve the boring problem of initialization / termination of library routines.
The problem is the following:

Initialization routine(s) must be executed before the main logic of a program can be started.
A typical use for an initialization routine is to create resources, such as critical sections, that will be required later by routines called ***in a multithreaded context***. Because of the multithreaded context, the resources must obviously be created and initialized before the threads that use them are created. Otherwise, a race condition could occur when two threads are trying at the same time to create a critical section, for instance. And it is a deadly sin to initialize a critical section twice…
So the only solution is to execute all the initialization routines at a time when the main thread of the process is the only one running. Of course, it is possible to call these routines explicitly from the startup code of your program. But for large programs, this is boring, error prone and often wasteful.

- Boring, because you have to look up each routine you use, check for an initialization routine and invoke it.
- Error prone, because if you forget to insert the initialization call in the startup code whenever you reference a new library routine, disaster will strike at some point. And then, later, during testing, you might figure out that you forgot to call the termination routine and some cleanup action never gets properly performed.
- Wasteful, because if one day, you stop invoking a given routine in your program, you will likely forget to remove the initialization and/or termination routine, and even if this does no harm, this will result in the linker still pulling the whole library member in your code, because the initialization code is still invoked there.

The solution to this are the four Init/Exit macros.

You code the $InitRoutine macro in the same module as the library routine itself, together with the initialization code for the routine. Ditto for the $ExitRoutine macro. So the library routine and everything related to its initialization and termination can be coded in the same source module, that of the runtime library, and only there. From them on, you don't have to think about initializing and terminating a library routine anymore.

If you call a library routine from within your code, this will pull its initialization code at the same time. And the initialization code of all library routines will be invoked all at once, in the startup code of your program, but you will not even have to know which routines actually require initialization and which do not.
Stop using some library routine, and it will not be pulled in the .EXE file anymore. Nor will its initialization code.

So where is the trick?
The trick is in the linker (and its "Grouped Section" feature mentioned above) and in four rather simple macros.
The macros create two sections. One is used to build a table containing the addresses of all those initialization routines. The other one is used to build a table containing the addresses of all the termination routines.

The $InitRoutine is used to declare an entry point to an initialization routine. Each $InitRoutine macro will define the address of its initialization routine and put it in the initialization section. Ditto for each $ExitRoutine, that will put the address of its exit routine in the termination section. If convenient, a single module might have as many $InitRoutine and $ExitRoutine macro calls as needed.

When the linker pulls a library routine, it will find their initialization and termination sections (if any) and concatenate them with those of the other library routines used by the program.

The only thing that the application startup code will need to do is to declare a $RunInitRoutines macro in its startup code, before any thread is created. The $RunInitRoutine does not take any parameter and does not even "know" whether there is any initialization routine to call in the whole project. If there is no initialization to perform, the initialization section will be empty.

The $RunInitRoutine generate a very simple loop that calls in turn each address in the initialization section.
Ditto for the $RunExitRoutines macro, that is called by the application in its final code, presumably after all active threads have terminated.

An example is probably appropriate at this point:

The modules requiring initialization and / or termination look like this:

In a module defining memory pools handling routines:

```
; Declare all routines that require initialization.

        $InitRoutine MPInitialize;Initialize memory pools
        $InitRoutine MPCritSectInit     ;Initialize critical section
                                                ;for the mempool routines.
        $ExitRoutine MPTerminate        ;Cleanup / release memory pools


MPInitialize PROC
                                                ; Initialization code here…
  RET
MPInitialize ENDP

MPCritSectInit PROC
                                                ; Initialization code here…
  RET
MPCritSectInit ENDP

MPTerminate PROC
                                                ; Initialization code here…
  RET
MPTerminate ENDP
```

The memory pool handling routines above need both initialization and termination.

In a module defining message queues handling routines:

```
; Declare the routine that requires initialization.

        $InitRoutine MQInitialize       ;Initialize message queue

MQInitialize PROC
                                                ; Initialization code here…
  RET
MQInitialize ENDP
```

The message queue routines above require an initialization routine but no termination routine.

The main code for a process using the initialization macros looks like this:

```
  .CODE

MainProc PROC

; At this point, no other thread is running, so none of the
; initialization routines encurs the risk of a race condition.

  $RunInitRoutines            ;Run all initialization routines.
                                    ;Carry will be set if some init
                                    ;routine failed. In this case,
                                    ;execution of init routines will
                                    ;stop after the failing routine.

    .IF !CARRY                ;If no initialization error,
    CALL MyMainCode           ;the main code is here…
    .ENDIF

  SAVE EAX
  $RunExitRoutines            ;Now execute all the registered
  RESTORE EAX                       ;Exit routines.

  INVOKE ExitProcess,         ;All done,
         EAX                  ;pass exit retcode.

MainProc ENDP
```

Here is the detailed, “under the hood” view of the four macros. You do not need to know exactly how this works to use it, anyway. If you don’t care, just skip to the next paragraph.

The $InitRoutine macro allows automatic, application-wide registering of the initialization routines.
The $ExitRoutine macro accomplishes the same for termination routines.

The $InitRoutine is used in any module containing initialization routine(s) that must be executed before the main logic of a program can be started.

$InitRoutine and $ExitRoutine are used to declare initialization and exit routines.
$InitRoutine declares a section, naming it @Init$<module name>.
$ExitRoutine declares a section, naming it @Exit$<module name>.
So if invoked in .ASM module FOO, $InitRoutine and $ExitRoutine will create segments named @Init$FOO and @Exit$FOO respectively. Both macros are called with the name of a routine. The macro will generate a DWORD pointer to the routine, and place this pointer in the @Init$ (or @Exit$) segment/section.
When a section name contains a '$' sign, a PE linker processes it specially, as mentionned in “Grouped Sections”, page 41

As a result, the contents of all "@Init$<modulename>" segments will be concatenated in the "@Init” section and the contents of all "@Exit$<modulename>" sections will be concatenated in the "@Exit" section. The linker sorts the sections fragments by alphabetical order of <modulename>.

So the @$InitRoutine macros of all modules contribute to the construction of a global table containing all the addresses of the initialization routines and located in section "@Init".
Likewise, the $ExitRoutine macros contribute to the construction of a global table containing all the addresses of the Exit routines and located in section "@Exit".

The $RunInitRoutines and $RunExitRoutines in the startup / exit code of the application put all of this together: they create the "@Init$" and "@Exit$" segments (that will end up ahead of all other @Init and @Exit segments in alphabetical collating sequence and contain the label at the top of the table), and "@Init$zzzzzzzz" / "@Exit$zzzzzzzz" (that will hopefully end up alphabetically after all other @Init and @Exit segments and contain a DWORD 0 as an end of table marker).

Finally, both the $RunInitRoutines and the $RunExitRoutines macro generate a short code loop that goes down its associated list and calls each address in the list.

In case the order of execution of some Init and/or Exit routines must be ordered differently, it is possible to pass a second (optional) parameter to the $InitRoutine ($ExitRoutine) declaration.
This second parameter is concatenated in the segment name ahead between the @Init$ (@Exit$) and before the <module name>. It allows one to change the linking order inside the group section and force the Init (Exit) routines to execute in any suitable order (rather than by alphabetical order of modules).
For instance, a "Console Log" routine might need initialization before any other routine so the other initialization routines might use the Console Log PROCs to log what they did. Passing a second parameter of "0" (lowest in collating sequence) might force the console log init routine to move ahead of the list (providing no other less important module uses this same value and no module is named "0.ASM").

## 5.4 The service routines

### 5.4.1 FatalError

See DEBUG.ASM.

# 6 Bibliography

## 6.1 [Booth, 96.01]

Rick Booth
Inner Loops
Addison Wesley Developer’s Press

## 6.2 [Brain, 96.01]

Marshall Brain

Win32 System Services (Second edition)
Prentice Hall PTR

## 6.3 [Intel, 95.01]

AP-526 Application Note
Optimizations For Intel's 32-Bit Processors
(available as electronic documentation at www.intel.com)

## 6.4 [Petzold 96.01]

Charles Petzold
Programming Windows 95
Microsoft Press

## 6.5 [Pietrek 95.01]

Matt Pietrek
Windows 95 Systems Programming Secrets
IDG Books

## 6.6 [Rector & al, 96.01]

Rector & Newcomer
Win32 Programming,
Addison Wesley Developers Press

## 6.7 [Richter 96.01]

Jeffrey Richter
Win32 Q & A
in Microsoft Systems Journal, Sept 96, Vol. 9

## 6.8 [Richter 97.01]

Jeffrey Richter
Advanced Windows (Third edition)
Microsoft Press

## 6.9 [Schulman 94.01]

Andrew Schulman
Unauthorized Windows 95
IDG Books