Instructions for objconv

A utility for cross-platform development of function libraries, for converting and modifying object files and for dumping and disassembling object and executable files for all x86 and x86-64 platforms.

Version 2.49. By Agner Fog © 2018.

GNU General Public License.

Contents

1 Introduction 2

  1.1 File types 3

2 Command line syntax 4

3 Warning and error control 6

4 Converting file formats 6

5 Modifying symbols 7

6 Managing libraries 8

7 Dumping files 10

8 Disassembling files 10

  8.1 How to interpret the disassembly 11

  8.2 Compatibility problems 13

  8.3 Using the disassembler for checking machine code 14

  8.4 Assembly syntax for AVX-512 and Knights Corner instructions 14

9 Converting assembler-generated files 16

10 Converting compiler-generated files 18

  10.1 Call stubs for 64-bit conversions 20

11 Frequently asked questions 22

  11.1 Why is there no graphical user interface? 22

  11.2 What kind of files can objconv convert? 22

  11.3 Is it possible to convert files for ARM? 23

  11.4 Is it possible to convert files for PPC or other architectures? 23

  11.5 Is it possible to link converted files into Borland Delphi Pascal? 23

  11.6 Can I convert an executable file from one system to another? 23

  11.7 Can I convert from 32 bit code to 64 bit code? 23

  11.8 Can I convert a dynamic link library to another system? 23

  11.9 Can I build a function library that works in all operating systems? 23

  11.10 Why can't I convert an export library? 23

  11.11 Can I convert a static library to a dynamic library? 24

  11.12 Can I convert a dynamic library to a static library? 24

  11.13 Can I convert a Windows function library to use it under Linux? 24

  11.14 Can I convert a Linux function library to use it under Windows? 24

  11.15 I want to know which library contains a particular function 24

  11.16 How do I know if my Linux function uses the red zone? 24

  11.17 How do I know if my Linux function has position-independent code 24

  11.18 I have problems porting my Windows application to Linux because the Gnu compiler has a more strict syntax. Can I convert the compiled Windows code instead? 25

  11.19 Is it possible to extract one or more functions from a binary file or program? 25

  11.20 Is it possible to convert mangled function names? 25

  11.21 Is it possible to convert function calling conventions automatically? 25

  11.22 Does the disassembler have an interactive feature? 25

  11.23 Is it possible to disassemble an executable file to modify it and then assemble it again? 25

  11.24 Is it possible to disassemble an object file and fix all compatibility problems manually? 26

  11.25 Is it possible to reconstruct C++ code from a disassembly? 26

  11.26 Why do I get error messages in the disassembly file? 26

  11.27 How does the disassembler distinguish between code and data? 26

11.28 Can I disassemble byte code? 27
11.29 Can I assemble the output of the disassembler? 27
11.30 Why does the disassembler not support AT&T syntax? 27
11.31 How can I convert assembly syntax? 27
11.32 Why does my disassembly take so long time? 27
11.33 How can I save the output of the dump screen to a file? 27
11.34 Can you help me with my problems? 27
11.35 Are there any alternatives to objconv? 28
12 Warning and error messages 28
12.1 Linker errors: 29
13 Source code 30
13.1 Explanation of the objconv source code 30
13.2 How to add support for new file formats 32
13.3 How to add features to the disassembler 32
13.4 File list 33
13.5 Class list 34
14 Legal notice 37

1 Introduction

Objconv is a utility for facilitating cross-platform development of function libraries, for converting and disassembling object files, and for other development purposes. The latest version of objconv is available at www.agner.org/optimize.

Objconv can perform the following tasks:

- Convert object files between different formats used on different x86 and x86-64 platforms.
- Change symbol names in object files.
- Build, manage and convert static link libraries in various formats for different x86 and x86-64 platforms.
- Dump file headers and other contents of object files, static and dynamic library files, and executable files.
- Disassemble object files and executable files and check instruction code syntax.

The following platforms are supported:

- Windows, 32 and 64 bit x86.
- Linux, 32 and 64 bit x86.
- BSD, 32 and 64 bit x86.
- Mac OS X, 32 and 64 bit x86 (Darwin, Intel based).

The latter three platforms are all based on the UNIX heritage. I will use "Unix" as a common name for Linux, BSD and Mac on x86 an x86-64 platforms in this manual.

The source code for objconv can be compiled and run under any of these platforms. The program is compatible with standard make utilities.

Note that objconv is intended for programming experts. It is far from fool proof, and you need to have a very good understanding of how compilers and linkers work in order to use this program. Please do not send your programming questions to me - you will not get any answer.

1.1 File types

An executable file is a file containing machine code that can be executed. This can be a program file or a dynamic link library, also called shared object. The name shared object is used only in Unix-like systems, such as Linux, BSD and Mac OS X.

An object file is an intermediate file used in the building of an executable file. It contains part of the code that will make up the final executable file. An object file usually contains cross-references to functions in other object files.

A static link library means a collection of object files. This is called a static linking library file in Windows terminology or an archive in Unix terminology. I prefer to use the name library because an archive can also mean a .zip or .tar file.

Objconv cannot modify or convert executable files, including dynamic link libraries or shared objects, but it can dump or disassemble such files.

The following table summarizes the type of operations that objconv can do on various file types:

File type and format	Word size, bits	Exten-sion	Operating system	Convert from	Convert to	Modify	Dump	Disas-semble
Object file COFF/PE	32, 64	.obj	Windows	x	x	x	x	x
Library file COFF/PE	32, 64	.lib	Windows	x	x	x	x	x
DLL, driver COFF/PE	32, 64	.dll, .sys	Windows	-	-	-	x	x
Executable file COFF/PE	32, 64	.exe	Windows	-	-	-	x	x
Object file OMF	16	.obj	DOS, Windows 3.x	-	-	-	x	x
Object file OMF	32	.obj	Windows	x	x	x	x	x
Library file OMF	16	.lib	DOS, Windows 3.x	-	-	x	x	x
Library file OMF	32	.lib	Windows	x	x	x	x	x
Executable file 16 bit	16	.exe	DOS, Windows 3.x	-	-	-	-	-
Object file ELF	32, 64	.o	Linux, BSD	x	x	x	x	x
Library file ELF	32, 64	.a	Linux, BSD	x	x	x	x	x
Shared Object ELF	32, 64	.so	Linux, BSD	-	-	-	x	x
Executable file ELF	32, 64		Linux, BSD	-	-	-	x	x
Object file Mach-O	32, 64	.o	Mac OS X	x	x	x	x	x
Library file Mach-O	32, 64	.a	Mac OS X	x	x	x	x	x
Shared object Mach-O	32, 64	.so	Mac OS X	-	-	-	x	x
Executable file Mach-O	32, 64		Mac OS X	-	-	-	x	x
Universal	32, 64		Mac OS X	-	-	-	x	x

binary								

2 Command line syntax

If you want to run objconv under one of the Unix systems (Linux, BSD, Mac), then you have to first build the executable. Unpack `source.zip` to a temporary directory and run the build script `build.sh`. To run objconv under Windows, you can just use the executable `objconv.exe`.

Objconv is executed from a command line or from a make utility. The syntax is as follows:

```
objconv options inputfile [outputfile]
```

Options start with a dash `-`. A slash `/` is accepted instead of `-` when running under Windows. Options must be separated by spaces. The order of the options is arbitrary, but all options must come before `inputfile`. The name of the output file must be different from the input file, except when adding object files to a library file. The option letters are case insensitive, file names and symbol names are case sensitive.

The return value from objconv is zero on success, and equal to the highest error number in case of error. This will stop a make utility in case of error messages, but not in case of warning messages.

Summary of options

`-fXXX`	Convert file to format `XXX`. `XXX` = `COFF`, `OMF`, `ELF` or `MAC`. `PE` is accepted as a synonym for `COFF`. The word size, 32 or 64, may be appended to the name, e.g. `ELF64`.
`-fasm`	Disassemble file. Variants for different assembly syntax dialects: `-fmasm`, `-ftasm`, `-fnasm`, `-fyasm`, `-fgasm`.
`-dXXX`	Dump contents of file. `XXX` can be one or more of the following: `f`: file header, `h`: section headers, `s`: symbol table, `r`: relocation table, `n`: string table (all names).
`-xs`	Strip exception handling information and other incompatible info. (Default when converting to a different format).
`-xp`	Preserve exception handling information and other incompatible info.
`-nu`	Change leading underscores on symbol names to the default for the target system.
`-nu-`	Remove leading underscores from symbol names.
`-nu+`	Add leading underscores to symbol names.
`-au-`	Remove leading underscores from public symbol names and keep old names as aliases.
`-au+`	Add leading underscores to public symbol names and keep old names as aliases.

`-nd`	Replace leading dot or underscore in section names with the default for the target system.
`-nr:N1:N2`	Replace name `N1` with `N2`. `N1` may be a symbol name, section name or library member name.
`-np:N1:N2`	Replace symbol prefix `N1` with `N2`. `N1` may be the beginning of a symbol name or section name.
`-ns:N1:N2`	Replace symbol suffix `N1` with `N2`. `N1` may be the end of a symbol name or section name.
`-ar:N1:N2`	Give public symbol `N1` an alias name `N2`. The same symbol will be accessible as `N1` as well as `N2`.
`-ap:N1:N2`	Replace symbol prefix `N1` with `N2` and retain the old name as an alias.
`-as:N1:N2`	Replace symbol suffix `N1` with `N2` and retain the old name as an alias.
`-nw:N1`	Make public symbol `N1` weak. Only possible for ELF files and 64-bit Mach-O files.
`-nl:N1`	Make public or external symbol `N1` local (invisible).
`-lx`	Extract all members from library `inputfile` to object files.
`-lx:N1:N2`	Extract member `N1` from library and save it as object file `N2`. The name of the object file will be `N1` if `N2` is omitted. May use `|` instead of `:` as separator.
`-la:N1:N2`	Add object file `N1` to library and give it member name `N2`. The member name will be `N1` if `N2` is omitted. May use `|` instead of `:`.
`-ld:N1`	Delete member `N1` from library.
`-ls`	Shorten long library member names. There are several different ways of storing member names longer than 15 characters in a library file. This option makes sure that no names are longer than 15 characters. This improves compatibility with all linkers, including BSD systems.
`-v0`	Silent operation. No output to console other than warning and error messages.
`-v1`	Verbose. Output basic information about file names and types (Default).
`-v2`	More verbose. Tell about conversions and library operations.
`-wdXXX`	Disable warning number `XXX`.
`-weXXX`	Treat warning number `XXX` as an error.
`-edXXX`	Disable error message number `XXX`.
`-ewXXX`	Treat error number `XXX` as warning.
`-imagebase=XXX`	Specify desired image-base as a hexadecimal number. (Only

	used if converting incompatible relocation types).
`@RFILE`	Read additional command line parameters from response file `RFILE`.
`-h`	Help. Print list of options.

Command line parameters can be stored in a response file. This can be useful if the command line is long and complicated. Just write `@` followed by the name of the response file. The contents of the response file will be inserted at the place of its name.

Response files can be nested, and there can be a maximum of ten response files.

Response files can have multiple lines and can contain comments. A comment starts with `#` or `//` and ends with a line break.

3 Warning and error control

Objconv can be called from a make utility. The make process will stop in case of an error message but not in case of warning messages. It is possible to disable specific error messages (`-edXXX`), to convert errors to warnings (`-ewXXX`) and to convert warnings to errors (`-weXXX`).

It is possible to disable error number 2005 is you want the input file and output file to have the same name. It is possible to disable error number 2505 if you want to mix object files with different word sizes in the same library.

4 Converting file formats

An object file can be converted from one format to another by specifying the desired format for the output file. The format of the input file is detected automatically. For example, to convert the 32-bit COFF file `file1.obj` to ELF:

```
objconv -felf32 -nu file1.obj file1.o
```

The name of the output file will be generated, if it is not specified, by replacing the extension of the input file with the default extension for the target format. The name of the output file must be different from the input file.

It is recommended to always use the `-nu` option. This makes objconv add or remove leading underscores on symbol names if required.

The output file will always have the same word size as the input file. It is not possible to change e.g. from 32-bit to 64-bit format.

A library is converted in the same way as an object file:

```
objconv -felf32 -nu file1.lib file1.a
```

Debug information and exception handling information is removed from the file, by default, if the format of the output file is different from the input file. It is recommended to remove this information because it will be incompatible with the target system. Objconv does not include a facility for converting this information to make it compatible.

Further instructions on converting assembler-generated and compiler-generated object code are given below in chapter 9 and 10.

5 Modifying symbols

It is possible to modify the names of public and external symbols in object files and libraries in order to prevent name clashes, to fix problems with different name mangling systems, etc.

Note that symbol names must be specified in the way they are represented in object files, possibly including underscores and name mangling information. All names are treated as case sensitive. Use the dump or disassembly feature to see the mangled symbol names.

To change the symbol name `name1` to `name2` in object file `file1.obj`:

```
objconv -nr:name1:name2 file1.obj file2.obj
```

The modified object file will be `file2.obj`. Objconv will replace `name1` with `name2` wherever it occurs in public, external and local symbols, as well as section names and library member names. All names are case sensitive.

It is possible to give a function more than one name. This can be useful for supporting multiple naming conventions with the same object or library file. Only public (exported) symbol names can have aliases. It is not possible to assign an alias to an external (imported) or local symbol. To give the function named `function1` the alias `function2`:

```
objconv -ar:function1:function2 file1.obj file2.obj
```

Some file formats have symbol names prefixed by an underscore (_) while other file formats have no prefix on symbol names. Use option `-nu` to change the prefix to the default for the target file format when converting from one format to another:

```
objconv -felf -nu file1.obj file2.o
```

Use option `-nu-` or `-nu+` to explicitly add or remove underscores on all symbol names.

You can specify any prefix to change or remove. For example, to remove prefix `_Win_` from all function names beginning with `_Win_`:

```
objconv -np:_Win_: file1.obj file2.obj
```

Likewise, you can modify all function names with a certain suffix. For example, to remove suffixes `@4`, `@8` and `@12` from all function names:

```
objconv -ns:@4: -ns:@8: -ns:@12: file1.obj file2.obj
```

You can keep the old names as aliases when modifying the prefix or suffix of function names. For example, to make a callable alias for Intel CPU-specific functions with suffix `.R`:

```
objconv -as:.R:_AVX: file1.obj file2.obj
```

No more than one operation can be specified for the same symbol name. For example, you cannot remove an underscore from a name and make an alias at the same time. You have to run objconv twice to so. For example, to convert COFF file `file1.obj` to ELF, remove underscores, and make an alias:

```
objconv -felf32 -nu file1.obj file1.o
objconv -na:function1:function2 file1.o file2.o
```

Likewise, you have to run objconv twice to make two aliases to the same symbol.

It is possible to make a public symbol weak in ELF and Mach-O files. A weak symbol has lower priority so that it will not be used if another public symbol with the same name is defined elsewhere. This can be useful for preventing name clashes if there is a risk that the same function is supplied in more than one library. Note that only the ELF and Mach-O file formats supports this feature. To make public symbol `function1` weak in ELF file `file1.o`:

```
objconv -nw:function1 file1.o file2.o
```

COFF and OMF files have a different feature called weak external symbols. This is not supported by objconv.

Objconv can hide public symbols by making them local. A public symbol can be made local if you want to prevent name clashes or make sure that the symbol is never accessed by any other module. To hide symbol `DontUseMe` in COFF file `file1.obj`:

```
objconv -nl:DontUseMe file1.obj file2.obj
```

It is also possible to hide external symbols. This can be used for preventing link errors with unresolved externals. The hidden external symbol will not be relocated. Note that it is dangerous to hide an external symbol unless you are certain that the symbol is never used. Any attempt to access the hidden symbol from a function in the same module will result in a serious runtime error.

All symbol modification options can be applied to libraries as well as to object files.

6 Managing libraries

A function library (archive) is a collection of object files. Each member (object file) in the library has a name which, by default, is the same as the name of the original object file.

All libraries contain a symbol index in order to make it easier for linkers to find out which member contains a particular function. Objconv will always remake the symbol index and remove the path from member filenames whenever a library file is modified.

`Objconv` can add, remove, replace, extract, modify or dump library members.

Rebuilding a library

Rebuilding a library will remove any path from member names, change the member name extension to `.obj` for COFF and OMF files, or `.o` for ELF and Mach-O files, and rebuild the symbol table. Example rebuilding library `mylib.lib`:

```
objconv mylib.lib mylib2.lib
```

Converting a library

To convert library `mylib.lib` from COFF to ELF format:

```
objconv -felf mylib.lib mylib.a
```

Building a library or adding members to a library

To add ELF object files `file1.o` and `file2.o` to library `mylib.a`:

```
objconv -la:file1.o -la:file2.o mylib.a
```

or alternatively:

```
objconv -lib mylib.a file1.o file2.o
```

The alternative `-lib` syntax is intended for `make` utilities that produce a list of object files separated by spaces. The library `mylib.a` will be created if it doesn't exist.

If you want to preserve the original library without the additions then give the new library a different name:

```
objconv -la:file1.o -la:file2.o mylib.a mylib2.a
```

Any members of the old library with the same names as the added object files will be replaced. Members with different names will be preserved in the library.

Any specified options for format conversion or symbol modification will be applied to the added members, but not to the old members of the library.

Removing members from a library

To delete member `file1.o` from library `mylib.a`:

```
objconv -ld:file1.o mylib.a mylib2.a
```

Extracting members from a library

To extract object file `file1.o` from library `mylib.a`:

```
objconv -lx:file1.o mylib.a
```

Any path of the original filename is ignored or removed by objconv. To extract library member `C:\some\very\long\path\file1.obj` from library `mylib.lib` and store it as `mypath\file1.obj:`

```
objconv -lx:file1.obj:mypath/file1.obj mylib.lib
```

You may use `|` instead of `:` as separator if the output path contains a colon:

```
objconv -lx|file1.obj|C:/mypath/file1.obj mylib.lib
```

To extract all object files from library `mylib.lib`:

```
objconv -lx mylib.lib
```

Any specified options for format conversion or symbol modification will be applied to the extracted members, but the library itself will be unchanged.

No more than one option can be specified for each library member. For example, you can't extract and delete the same member in one operation.

Modifying library members

To rename library member `file1.o` to `file2.o` in library `mylib.a`:

```
objconv -nr:file1.o:file2.o mylib.a mylib2.a
```

To rename symbol `function1` to `function2` in library `mylib.a`:

```
objconv -nr:function1:function2 mylib.a mylib2.a
```

Any symbol modification option specified will be applied to all library members that have a symbol with the specified name.

Dumping library contents

To show all members and their public symbol names in library `mylib.a`:

```
objconv -d mylib.a
```

Note that the member names shown are the names before conversion. All other commands use the member names after any path has been removed. See section 11.15 for how to list the contents of multiple libraries.

To show the complete symbol list of member `file1.o` in library `mylib.a`:

```
objconv -dhs -lx:file1.o mylib.a
```

To show all symbols in all members of library `mylib.a`:

```
objconv -dhs -lx mylib.a
```

7 Dumping files

Objconv can dump file headers, symbol tables, etc. for various types of files. For example, to dump the file header, section headers and symbol table of `file1.obj`:

```
objconv -dfhs file1.obj
```

8 Disassembling files

Objconv can disassemble object files, executable files, etc. For example, to disassemble the dynamic link library `file1.dll` to NASM syntax:

```
objconv -fnasm file1.dll file1.asm
```

To disassemble a static library file (`*.lib`, `*.a`) you must first extract the individual library members and then disassemble each member separately.

Three different syntax dialects are supported:

1. MASM/TASM. Used by Microsoft and Borland assemblers. This is the most common syntax used in manuals etc. Windows compilers can generate output in this format. Command line option `-fasm` or `-fmasm` or `-ftasm`.

2. GAS. Used by the Gnu compiler and assembler. Only the Intel syntax sub-version is supported. Use this for inline assembly with the gcc or g++ compiler. Command line option `-fgasm`.

3. NASM/YASM. Used by NASM and YASM. These are free assemblers with support for multiple platforms. This syntax is more logical and consistent than the other dialects, but with fewer options. Command line option `-fnasm` or `-fyasm`.

The output file is written in such a way that it can be assembled again with the appropriate assembler. Possible problems with re-assembling the file are discussed below.

The disassembler supports the full instruction set for all 16-, 32- and 64-bit x86 Intel, AMD and VIA processors, including the Intel SSE, AVX, AVX2, AVX512F/VL/BW/DQ/CD/IFMA/VBMI, FMA3, BMI1, BMI2, etc., AMD XOP, FMA4 and TBM instructions, VIA instructions, privileged instructions, the Intel Knights Corner instruction set, known undocumented instructions, and preliminary instruction codes that were never implemented because of changed plans (e.g. SSE5), totaling approximately 2000 instructions.

The quality of the disassembly depends on the amount of information contained in the input file. Object files generally contain more information about symbol names, types, etc. than executable files do. COFF and ELF files contain more symbol names than OMF and Mach-O files do.

The disassembler analyzes the code in order to determine the type of each data item, to guess where each function begins and ends, to identify import tables, switch/case jump tables, virtual function tables, etc. Nevertheless, the disassembler may in difficult cases misinterpret data as code or fail to determine the type of a data item. When the disassembler is in doubt whether something is code or data, it will show it as both.

In simple cases, the quality of the disassembly may be good enough for making modifications in an object file or for extracting a single function from a dynamic link library. The disassembly of an executable file is unlikely to be good enough for remaking a fully working executable, but it may be good enough for identifying problems in the code.

8.1 How to interpret the disassembly

The following example shows what a piece of disassembled code may look like (32-bit Windows, MASM syntax):

```
_text   SEGMENT PARA PUBLIC 'CODE'                      ; section number 1

?testb@@YAHH@Z PROC NEAR
        mov     eax, dword ptr [esp + 04H]              ; 0000 _ 8B. 44 24, 04
; Note: Memory operand is misaligned
        mov     ecx, dword ptr [?alpha@@3HA]            ; 0004 _ 8B. 0D, 00000000(d)
        add     ecx, eax                                ; 000A _ 03. C8
        push    ecx                                     ; 000C _ 51
        call    ?testa@@YAHH@Z                          ; 000D _ E8, 00000000(rel)
        add     esp, 4                                  ; 0012 _ 83. C4, 04
        mov     ecx, offset ?list1@@3PAHA               ; 0015 _ B9, 00000000(d)
; Filling space: 06H
; Filler type: lea with same source and destination
;       db 8DH, 9BH, 00H, 00H, 00H, 00H
ALIGN   8
?_001:  add     eax, dword ptr [ecx]                    ; 0020 _ 03. 01
        add     ecx, 4                                  ; 0022 _ 83. C1, 04
        cmp     ecx, offset ?list1@@3PAHA + 00001000H   ; 0025 _ 81. F9, 00001000(d)
        jl      ?_001                                   ; 002B _ 7C, F3
        ret                                             ; 002D _ C3
```

```
?testb@@YAHH@Z ENDP
_text   ENDS
```

This code can be interpreted as follows:

The name `?testb@@YAHH@Z` is the name of the function `int testb(int x)` as it is mangled by the Microsoft C++ compiler. The disassembler does not translate mangled names to C++ names for you. The MASM assembler allows the characters `? @ $ _` in symbol names.

Line `0000` is the first instruction of the function `testb`. It reads the parameter `x` from the stack into register `eax`. Line `0004` reads a value from a variable in the data segment into `ecx`. The name `?alpha@@3HA` is a mangled name for `int alpha`. The note indicates that `alpha` is not optimally aligned. Such notes always apply to the instruction that follows. Line `000A` adds the value of `x` in `eax` to the value of `alpha` in `ecx`. Line `000C` pushes this value on the stack as a parameter to the following function call. Line `000D` is a call to function `int testa(int)` with a mangled name. The return value is in `eax`. Line `0012` cleans up the stack after the function call. Line `0015` loads the address of `?list1@@3PAHA` into `ecx`. This is the mangled name of an array `int list1[]`.

Next comes a multi-byte `nop` for aligning the subsequent loop entry. The compiler has used `lea ebx,[ebx+00000000H]` instead of 6 `nop` instructions for filling 6 bytes. The disassembler has written the exact byte sequence as a comment. This may be uncommented to recover exactly the same code, but in general it is preferred to use the `align` directive instead. The disassembler cannot know whether the desired alignment is 8 or 16 if there are less than 8 bytes up to the next 16-bytes boundary.

Line `0020` is a loop entry with the label `?_001`. The input file does not indicate a name for this label. Therefore the disassembler has assigned the arbitrary name `?_001`. Subsequent nameless code and data labels will be named `?_002`, etc.

The first line in the loop reads an integer from the address that `ecx` points to, i.e. an element from array `list1`, and adds it to `eax`. Line `0022` adds 4, which is the size of each array element, to `ecx` in order to make it point to the next array element.

Line `0025` compares `ecx` with the address of the end of the array. Line `002B` reads the flags from the preceding `cmp` instruction and jumps back to the top of the loop if the end of the array has not been reached. Line `002D` returns from function `testb`. The return value is in `eax`.

This code could be translated back to C++:

```
int testa(int x);
int list1[1024];
int alpha;

int testb(int x) {
   int y = testa(x + alpha);
   for (int i=0; i<1024; i++) y += list1[i];
   return y;
}
```

The comments to the right of the disassembly code are interpreted as follows. The four digits after the semicolon is the hexadecimal address of the instruction. This is actually a 32-bit value, but in this case the disassembler has saved some space by using only 4 hexadecimal digits. It will show 8 hexadecimal digits if necessary, but not more. Addresses higher than 2^{32} will be shown only as the least significant 8 hexadecimal digits.

After the underscore comes the instruction code as hexadecimal bytes. The delimiters `: . ,` separate the different parts of the instruction code.

The text in parenthesis after the binary code indicates various types of cross-references, using the following abbreviations:

Abbreviation	Cross reference type
d	Direct address. The absolute virtual address of target is inserted
rel	Self-relative address
imgrel	Image-relative address
segrel	Address is relative to a segment or group
refpoint	Address is relative to an arbitrary reference point
indirect	To Gnu indirect function dispatcher
seg	A segment address or segment descriptor
sseg	Only the segment part of a far target address is inserted
far	Offset and segment of a far target address
GOT	Global offset table entry
GOT r	Self-relative address of global offset table entry
PLT r	Self-relative address of procedure linkage table entry

The information about cross-reference types is usually obtained from relocation tables in the input file. The disassembler will attempt to reconstruct missing cross-reference information, if possible, in the case of executable files without relocation tables.

8.2 Compatibility problems

Even though the goal has been to make the disassembly output fully compatible with the specified assembler, there are still some possible compatibility problems. The following types of problems may occur when re-assembling disassembled code:

- Unsupported relocation types. The original file may contain relocation types not supported by the assembler. Image-relative relocations are supported only by MASM. Relocations relative to an arbitrary reference point are supported only by the Macintosh version of the Gnu assembler (which currently doesn't support the Intel syntax variant). Relocations to a global offset table (GOT), procedure linkage table (PLT) or other import tables are only partially supported by the disassembler. The type of relocation is indicated in the comment only, not in the instruction. The GOT, PLT, import tables, etc. are shown as data if contained in the input file.

- Nonstandard segment names. Most assemblers have little or no support for code segments with nonstandard names.

- Nonstandard segment attributes. Most assemblers have little or no support for specifying segment attributes such as executable, writeable, zerofill, etc.

- Nonstandard segment alignment. MASM sets the alignment for _text and _data to 16 in 64 bit mode or if `.xmm` is specified, and 4 if `.xmm` is not specified. If the default alignment does not fit your purpose then append a `$`-sign and something to the segment name, e.g. `_text$align32` and specify the desired alignment.

- Special characters in function names. The following special characters are allowed in identifiers: NASM/YASM: `_$@?.~#`, Gas: `_.$`, MASM: `_$@?.` ('`.`' only in the beginning of a name). The disassembler will count names containing illegal characters and write a notice in the beginning of the file.

- Exception handling information and debugging information. This information is shown only as data. The appropriate directives are not inserted in the code. Use option `-xs` to remove exception handling and debugging information.

- Communal code and data. This will be converted to public when re-assembled. A comment is inserted in the disassembly file indicating communal code or data.

- Newer instruction sets. The disassembler supports the newest instruction sets currently available. The assembler may not support the same instruction sets. The NASM assembler is often the first to support new instruction sets.

- Executable files. Executable files and dynamic link libraries or shared objects contain import tables and other information that will not survive a disassembly and re-assembly. It may be possible to recover individual functions from an executable file but not the entire program.

8.3 Using the disassembler for checking machine code

The disassembler does an almost complete syntax check of the code. This can be useful for debugging purposes and for testing compilers and assemblers during development. For example, it will write an error message in the output file if there is a memory operand on an instruction that allows only register operands. Less serious errors, such as redundant prefixes, are written as "Note" rather than "Error".

The disassembler also checks for some cases of suboptimal code, for example unaligned memory operands, length-changing prefixes, and instructions that could have been coded in a shorter form.

The disassembler does not check for programming errors, such as for example a `push` that doesn't have a matching `pop`.

A note or error message does not necessarily indicate an error in the compiler that built the code. Compilers may sometimes have good reasons for coding an instruction in an apparently suboptimal form. Error messages typically occur when the compiler has placed data in a code segment and the disassembler has failed to identify this as data. Another possible cause of errors is misplaced labels caused by address calculations that the disassembler has failed to trace correctly. It is very unlikely that the error messages you see are caused by bugs in the compiler.

8.4 Assembly syntax for AVX-512 and Knights Corner instructions

The disassembler supports the instruction set for the AVX-512 instructions and the instruction set for Intel "Many Integrated Core" (MIC) coprocessor codenamed Knight's Corner. See Intel manuals. These two instruction sets are very similar, but have different optional instruction attributes. Instructions from these two instruction sets differ by a single bit in the prefix, even for otherwise identical instructions.

These instruction sets extend the size of vector registers to 512 bits. The number of vector registers is extended to 32 vector registers named zmm0 - zmm31 in 64-bit mode. Only zmm0 - zmm7 are available in 32-bit mode. The new instructions have many new attributes for masked operations, broadcast, rounding mode, suppression of exceptions, type conversion, permutation, and cache eviction hint.

These instruction sets are not yet supported by all assemblers (December 2014), and the assembly syntax details have only been defined for the NASM assembler. It is therefore useful to specify the used syntax here. The syntax described below is used in the disassembler.

512 bit memory operand size specifier: MASM and GAS syntax: `zmmword`, NASM syntax: `zword`.

Masked operation: `{kn}`, where `kn` = `k1`, `k2`, ... `k7` is the mask register. This attribute is written after the destination operand. This may be omitted for `{k0}`. The disassembler writes `{k0}` explicitly only if the k0 register is modified by the instruction. A mask register used for other purposes is written like a normal operand without curly brackets.

Broadcast for memory operand: `{1to8}` etc. Written after the memory source operand.

Rounding mode: `{rn}` etc. Written after a comma after the last SIMD operand.

Suppress all floating point exceptions: `{sae}`. Written after a comma after the last SIMD operand.

Rounding mode and `sae` may optionally be combined: `{rn-sae}`.

The AVX-512 and Knights Corner instructions apply a multiplier to the address offset of memory operands with a pointer register and a one-byte offset. This multiplier is usually the same as the actual size of the source operand before any broadcast or conversion or the destination operand after any conversion, with masks ignored. The disassembler writes the total offset as the product of the offset byte and the multiplier to show how the value is calculated, for example:

```
vaddps zmm1 {k2}, zmm3, dword [rsi+12H*4H] {1to16}
```

An assembler should accept the total offset as well (e.g. `[rsi+48H]`) and use a 32-bit offset without multiplier in case the specified offset is not divisible by the multiplier.

Attributes available only with AVX-512 instructions:

Zeroing: `{z}` written after the destination register and after the mask specifier.

Attributes available only with Knights Corner instructions:

Cache eviction hint: `{eh}`. Written after the memory operand.

Type conversion: `{uint16}` etc. Written after the source or destination memory operand. Note that the specified operand size applies to the actual size of the converted memory operand with masks ignored.

Broadcast for register operand: `{aaaa}` etc. Written after the register source operand.

Permutation (swizzle): `{cdab}` etc. Written after the register source operand.

An extra comma is inserted only between the last operand, and attributes that do not apply to a specific operand, i.e. rounding mode and suppress-all-exceptions.

Multiple attributes on the same operand are written in separate curly brackets, for example:

```
vaddpd zmm30 {k3}{z}, zmm10, zmm8      ; AVX512 instruction

vmovdqa32 yword [rdi] {k1} {sint16} {eh}, zmm2 ; KNC instr.
```

Rounding mode and suppress-all-exceptions may be considered separate attributes written in separate curly brackets, or one combined attribute. For example:

```
vaddps  zmm10 {k4}, zmm20, zmm30, {rn} {sae}
```

or

```
vaddps  zmm10 {k4}, zmm20, zmm30, {rn-sae}
```

The disassembler currently uses the combined syntax.

9 Converting assembler-generated files

Objconv makes it possible to develop multi-platform function libraries from a single development platform. The code can be compiled or assembled on one platform and the resulting object or library files can then be converted to different file formats for different platforms.

It is preferred to make static libraries (`*.lib`, `*.a`) rather than dynamic link libraries or shared objects (`*.dll`, `*.so`). Shared objects in Unix systems require position-independent code that can cause compatibility problems.

It is recommended to use assembly code rather than C or C++ in order to avoid any platform-specific or compiler-specific constructs. Things that can go wrong when converting compiler-generated code are summarized on page 18 below.

The differences in calling conventions etc. are described in detail in my manual 5: "Calling conventions for different C++ compilers and operating systems". www.agner.org/optimize.

My manual 2: "Optimizing subroutines in assembly language" explains how to make function libraries that are compatible with multiple platforms. (www.agner.org/optimize).

32-bit code

The calling conventions and register usage conventions are the same on all 32-bit x86 platforms. This makes it easy to use the same code on different platforms. Differences that have to be dealt with are:

- Underscore prefixes. Function names and variable names get an underscore prefix in 32-bit COFF, OMF, and MachO files, but not in ELF. Objconv will automatically add or remove underscores, as required, with the `-nu` option.

- Function calling convention. The most common calling convention in 32-bit mode is `__cdecl`. Windows DDL's also use `__stdcall`. Some Windows compilers use `__thiscall` for class member functions. You can override the default __thiscall by specifying `__cdecl` in the definition of class member functions. Use `__cdecl` everywhere to prevent incompatibilities.

- Virtual functions, constructors, destructors, dynamic memory allocation, runtime type identification, structured exception handling, thread-local storage and other advanced C++ constructs should be avoided or tested thoroughly before you rely on it in converted code.

- Name mangling. Different compilers use different name mangling schemes for function names. This problem is usually dealt with by declaring all functions and shared global variables `extern "C"`. For example:

  ```
  extern "C" int function1(int x);
  extern "C" {int globalvariable1 = 0;}
  ```

Class member functions, overloaded functions and operators cannot be declared `extern "C"`. The mangled function names must be converted manually with the objconv `-nr` or `-ar` option. Alternatively, you may provide multiple mangled names for the same function in the assembly code:

```
name1 proc  near
name2 label near
name3 label near
public name1, name2, name3
; function body
ret
name1 endp
```

Another way to avoid name mangling is to define a mangled function that is replaced inline by a call to an unmangled function. See my manual 2: "Optimizing subroutines in assembly language" for examples. The different name mangling schemes are described in my manual 5 on calling conventions.

64-bit code

In 64-bit code we must take the same considerations as for 32-bit code. Function names have an underscore prefix only in 64-bit MachO files, not in COFF and ELF. However, there are several other issues to take care of when converting 64-bit code.

The calling conventions and register usage conventions in 64-bit Windows are different from the conventions in 64-bit Unix systems. You can support both sets of conventions by making functions with multiple entries in the assembly code. For example:

```
; C function prototype:
; extern "C" int Examplefunction (int a, double b);
; Assembly:
Unix_Examplefunction proc      ; Unix entry
    mov     ecx, edi           ; Unix: a=edi,  Windows: a=ecx
    movapd xmm1,xmm0           ; Unix: b=xmm0, Windows: b=xmm1
Win_Examplefunction label near ; Windows entry
    ; function body (a=ecx, b=xmm1)
    ret
Unix_Examplefunction endp
```

If we put parameter `b` before `a` then both systems will have `b` in `xmm0`, while `a` will be in `edi` and `edx`, respectively:

```
; C function prototype:
; extern "C" int Examplefunction (double b, int a);
; Assembly:
Unix_Examplefunction proc      ; Unix entry
    mov     edx, edi           ; Unix: a=edi,  Windows: a=edx
Win_Examplefunction label near ; Windows entry
    ; function body (a=edx, b=xmm0)
    ret
Unix_Examplefunction endp
```

I have chosen to put a `Win_` prefix on the Windows function entries and `Unix_` on the Unix function entries. It is easy to make objconv remove the `Win_` prefixes for the COFF files and remove the `Unix_` prefixes for the ELF and MachO files when converting the object or library file:

```
objconv -fcof64 -nu -np:Win_::   example.lib examplecof.lib
objconv -felf64 -nu -np:Unix_::  example.lib exampleelf.a
objconv -fmac64 -nu -np:Unix_:_: example.lib examplemac.a
```

We must take care of the differences in register usage conventions. Registers `RAX`, `RCX`, `RDX`, `R8` - `R11` and `XMM0` - `XMM5` can be used without saving in both systems. Register `RSI`, `RDI` and `XMM6` - `XMM15` can be used without saving in Unix, but not in Windows. Register `RBX`, `RBP` and `R12` - `R15` must be saved and restored if used in both systems. To make the code compatible with both systems we must follow the strictest rule, which is the Windows rule. A function with two entries, as in the example above, must have the Unix entry before the Windows entry in order to avoid polluting register `RSI` and `RDI` with the function parameters used by Unix when calling from Windows.

If the function calls any other function which could possibly have the Unix convention, then we cannot rely on register `RSI`, `RDI` and `XMM6` - `XMM15` to be unchanged across the call to the other function.

Both sets of conventions have the stack aligned by 16 before every `CALL` instruction. The Windows convention dictates that 32 bytes of "shadow space" must be allocated on the stack before a function call. This shadow space belongs to the called function. Unix does not have the shadow space. A function compatible with both sets of conventions should not use any shadow space, but must allocate a shadow space to any function it calls which possibly could have the Windows convention.

The Unix convention allows functions to use the "red zone" of 128 bytes above the stack, while Windows does not have a red zone. Avoid using the red zone in multi-platform functions.

If the multi-platform function calls another multi-platform function then we can of course rely on the latter function to conform to the strictest convention, but if we are calling a standard library function, which is available on both platforms, then we must take all of the above precautions.

10 Converting compiler-generated files

It is always risky to convert compiler-generated object and library files to a different file format because the compiler might link to other functions or data that are not available on the target system. If the source code is available then, by all means, you should prefer to recompile the code on the target platform rather than convert the compiled code. Any problems you may encounter because of differences in C++ syntax are small compared to the problems of incompatibilities in the binary interface.

If the source code is not available then you may try to convert the object code. It works in simple cases, but be prepared for a lot of problems if the function contains incompatible structures or accesses other incompatible functions or data.

Another possibility is to disassemble the object code, fix all compatibility problems manually, and then assemble again. This may be the only solution in some cases, but it requires a lot of experience to understand the disassembled code. A disassembly or file dump may also be helpful for diagnosing conversion problems.

The following table summarizes the main reasons why converted object code may fail.

Reasons why conversion of compiler-generated code may fail	
Compiler-specific library calls	Most compilers can generate calls to library functions that are specific to that particular compiler or use compiler-specific global variables. It

	may be necessary to convert the called functions as well or make replacements for the missing functions or variables
Calls to operating system	Operating system calls are not compatible among systems.
Calling conventions in 32-bit mode	Most compilers support the same calling conventions in 32-bit mode. You may have to specify a specific calling convention, preferably `__cdecl` as explained above.
Calling conventions in 64-bit mode	The calling conventions in 64-bit Windows and 64-bit Unix systems are different. You need a call stub as explained below.
Register usage conventions in 32-bit mode	The register usage conventions are the same in all 32-bit systems, except for Watcom compilers.
Register usage conventions in 64-bit mode	Linux functions may modify registers `RSI`, `RDI` and `XMM6` - `XMM15`, which must be preserved by Windows functions. You need a call stub to fix this incompatibility.
Red zone	64-bit Unix systems allow functions to use a "red zone" of 128 bytes above the stack for local storage. Windows does not specify a red zone. If a converted Unix function uses the red zone under Windows it will usually work. The Windows system will switch stacks in case of an interrupt so the red zone is not overwritten, but the system could possibly discard the red zone if it is low on memory. This could produce extremely rare and irreproducible errors. No error will happen if the Unix function is compiled with option `-mno-red-zone`.
Leading under-scores on names	Use the `-nu` option on objconv to add or remove leading underscores as needed.
Mangling of function names	Different compilers use different name mangling schemes. Use `extern "C"` on all function declarations in C++ to avoid name mangling. If this is not possible then you may have to change the mangled name by using the `-nr` option in objconv.
Initialization and termination code	Initialization and termination code is used for calling the constructors and destructors of global objects and for initializing function libraries, etc. Objconv attempts to convert the initialization code, but the termination code is often incompatible and will not work.
Exception handling and stack unwinding information	This information is not compatible between different systems. Objconv will remove this information by default. Do not rely on structured exception handling. Do not rely on destructors being called at `longjmp` or when a thread is terminated.
Other advanced C++ constructs	Virtual functions, constructors, destructors, dynamic memory allocation, runtime type identification, thread-local storage, member pointers and other advanced C++ constructs may not work after conversion.
Communal functions and data	Objconv does not include a feature for converting communal (coalesced) data. Do not rely on function-level linking (`/Gy`) on Microsoft compilers or `-ffunction-sections` on Gnu compilers. Communal functions will be converted to non-communal in some cases. Conversion of communal functions in OMF files is not supported.
Incompatible relocation types	Mach-O files allow a relocation type that computes addresses relative to an arbitrary reference point. This is not supported by other systems. 64-bit COFF files may contain image-relative relocations not supported in ELF. 64-bit ELF files may contain 32-bit absolute addresses not supported in Mach-O. Objconv may be able to work around some of these problems if a specific image base is specified.
Position-independent code	Unix systems require position-independent code when making shared objects (*.so). Windows compilers are not able to make position-independent code. Use static linking when using converted object files on these systems. Avoid conversion of compiler-generated position-independent code (use g++ option `-fno-pic`). See my manual

	"Optimizing subroutines in assembly language" for instructions on how to make position-independent 32-bit code in assembly.
Lazy binding	Import tables for lazy binding of external references are not compatible between different systems. Objconv will convert lazy to non-lazy references in some cases.
Default library information	Information in object files about which libraries to include is not converted by objconv because the libraries are unlikely to have the same names in the target system.

Conversion between different Unix systems is more likely to be successful than conversion between Unix and Windows.

10.1 Call stubs for 64-bit conversions

It is necessary to use call stubs when converting 64-bit compiler-generated code between Windows and Linux systems. Call stubs are not needed for 32-bit code or when converting between different Unix systems.

The purpose of the call stubs is to take care of the differences in calling conventions and register usage conventions between 64-bit Windows and 64-bit Unix. Several standard call stubs are provided with objconv: `w2ustub.o` is needed when calling a 64-bit function that has been converted from Windows to Unix. `u2wstub.obj` is needed when calling a 64-bit function that has been converted from Unix to Windows. You can find these files in `extras.zip`.

The standard stubs `w2ustub.o` and `u2wstub.obj` work only when the converted function satisfies the following conditions:

- The function must have no more than four parameters.
- The parameters cannot be a composite type (`struct`, `class`), but pointers and references to such types are allowed. Member pointers are not allowed. Arrays of any type are allowed.
- If any parameter is of type `float` or `double` then there can be no parameters of any other type than `float` and `double`. `long double` cannot be used.
- Parameters of intrinsic vector types (`__m128`, `__m128d`, `__m128i`) require a different stub, see below.
- The function cannot have a variable parameter list, such as `printf`.
- The return can be `void` or any type. If the return is a composite type then this may use a return pointer, counting as one parameter. Class member functions have an implicit `this` pointer, also counting as one parameter.
- No stub is needed in 32-bit mode. No stub is needed when converting between Linux, BSD and Mac.

If these conditions are not met, i.e. if the function has more than four parameters or if it has a mixture of floating point and integer parameters, then you have to make a tailor-made call stub in assembly language. See the source code `w2ustub.asm` and `u2wstub.asm`.

The following examples explain how to use the call stubs. Assume that you have a 64-bit Windows function library containing a function called `Alpha` that you want to use in a Linux system. For example, `Alpha` can have the following definition:

```
extern "C" int Alpha(int a, int b);
```

A dump of the library shows that the function `Alpha` is in module `alpha.obj`. We will extract this from the library:

```
objconv -fcof64 -lx:alpha.obj somelibrary.lib
```

Now we want to convert `alpha.obj` to ELF format. At the same time we can change the name of function `Alpha` to something else, e.g. `w_Alpha`:

```
objconv -felf64 -nr:Alpha:w_Alpha alpha.obj alpha.o
```

The reason why we want to change the name is that we want to call the function through a call stub. The main program calls a stub named `Alpha`, which in turn calls the converted function `w_Alpha`.

Now we can make the stub from `w2ustub.o` by inserting the names in this standard stub:

```
objconv -felf -nr:uname:Alpha -nr:wname:w_Alpha w2ustub.o astub.o
```

We can now build the executable from the main file, the converted object file and the stub. If the file `main.cpp` contains the call to `Alpha`:

```
g++ -m64 main.cpp alpha.o astub.o
```

We have to check if the converted function calls any other functions. Use the dump feature of objconv and look at the list of external symbols to see if the function needs access to other functions.

If the converted function `Alpha` contains a call to another Windows function, `Beta`, then the latter function must be converted as well. No stub is needed when a converted function `Alpha` calls another converted function `Beta`.

But if the converted Windows function calls a Unix function then this call must go through a reverse stub. Assume that the converted Windows function `Alpha` calls the standard library function `sin`. This function is available with the same name in both Windows and Unix libraries. Rather than converting the Windows math library (which would probably fail), we prefer to call the `sin` function in the Unix function library. We have to change the name of `sin` in `alpha.o` in order to avoid calling the Unix function library directly:

```
objconv -felf -nr:Alpha:w_Alpha -nr:sin:w_sin alpha.obj alpha.o
```

The reverse stub to call the Unix function `sin` from `Alpha` is made from `u2wstub.obj`:

```
objconv -felf -nr:uname:sin -nr:wname:w_sin u2wstub.obj sinstub.o
```

The final executable must include both the forward stub to call Windows function `Alpha` from Unix and the reverse stub to call Unix function `sin` from `Alpha`:

```
g++ -m64 main.cpp alpha.o astub.o sinstub.o
```

Calling a 64-bit Windows function from BSD goes in exactly the same way. In Mac systems we need underscore prefixes on the function names `_Alpha` and `_sin`:

```
objconv -fmac -nr:Alpha:w_Alpha -nr:sin:w_sin alpha.obj alpha.o
objconv -fmac -nr:uname:_Alpha -nr:wname:w_Alpha w2ustub.o astub.o
objconv -fmac -nr:uname:_sin -nr:wname:w_sin u2wstub.obj sinstub.o
```

```
 g++ -m64 main.cpp alpha.o astub.o sinstub.o
```

If we want to use a 64-bit Unix function in a Windows program, we can follow an analogous procedure with the stubs going in the opposite directions.

Assume that we have a 64-bit Linux function `Gamma` that we want to call from Windows. `Gamma` calls the standard library function `cos`, which is available in our Windows function library. First we convert the object file `gamma.o` to COFF format and change the function names in the file in order to insert call stubs:

```
objconv -fcof64 -nr:Gamma:u_Gamma -nr:cos:u_cos gamma.o gamma.obj
```

If the functions `Gamma` and `cos` satisfy the conditions for using the standard call stubs, then we can insert the names in the stubs:

```
objconv -fcof -nr:uname:u_Gamma -nr:wname:Gamma u2wstub.obj gstub.obj
objconv -fcof -nr:uname:u_cos -nr:wname:cos w2ustub.o cosstub.obj
```

Now we can insert the converted object file and the two stubs in the final executable:

```
cl main.cpp gamma.obj gstub.obj cosstub.obj
```

Converting from 64-bit MachO to COFF goes the same way, except for the extra underscores in the conversion:

```
objconv -fcof64 -nr:_Gamma:u_Gamma -nr:_cos:u_cos gamma.o gamma.obj
```

Special call stubs for functions with intrinsic vector parameters of type `__m128`, `__m128d` and `__m128i` are also provided. Use `w2ustubvec.o` for functions converted from Windows to Unix with 1 - 4 parameters of these types and no parameters of any other type. Use `u2wstubvec1.obj` or `u2wstubvec2.obj` for functions converted from Unix to Windows with exactly one or two parameters respectively of these types and no parameters of any other type.

More details about incompatibilities between different platforms are documented in my manual number 5: "Calling conventions for different C++ compilers and operating systems". (www.agner.org/optimize).

11 Frequently asked questions

11.1 Why is there no graphical user interface?

Most users will prefer to call objconv from a make utility, a script or a batch file. A graphical user interface would compromise the cross-platform portability of the source code.

11.2 What kind of files can objconv convert?

Objconv can convert object files (*.obj, *.o) and static library files (*.lib, *.a) for 32-bit and 64-bit x86 systems, such as Windows, Linux, BSD and Intel-based Mac OS X.

The conversion is most likely to be successful if the file is built from assembly code with careful consideration of the calling conventions etc. of the target system. Conversion of compiler-generated code works in simple cases where there are no system calls or other features known to cause problems. Conversion of 64-bit compiler-generated code between Windows and Unix systems works only if call stubs are inserted.

Se page 18 for a list of reasons why conversions may fail.

11.3 Is it possible to convert files for ARM?

No. A lot of people have asked about this, so there is obviously a need for such a tool, but I am not gonna make it. Objconv supports only files for x86 and x86-64 architectures. It will require a major rewrite of objconv to make a converter for ARM files. I don't know if other tools such as Gnu objcopy can do the job. If anybody out there has more information on this then please let me know so that I can put it into this FAQ.

11.4 Is it possible to convert files for PPC or other architectures?

No. Objconv supports only files for x86 and x86-64 architectures. It will require a major rewrite of objconv to make a converter for PPC files and there is a little/big endian issue to take care of.

11.5 Is it possible to link converted files into Borland Delphi Pascal?

Yes. The Turbo Delphi compiler accepts object files in 32-bit OMF format, but the object files must meet several requirements that are poorly documented: (1) The file cannot contain communal functions (also called function-level linking). Turn off this option in the compiler (e.g. on bcc32 compiler, use option `-VA-`). (2) All section names must begin with an underscore, not a dot. You can fix the underscores with `objconv -fomf -nu -nd inputfile.obj outputfile.obj`. (3) The object file must contain both `_text`, `_data` and `_bss` segments. If it doesn't, then add at least one initialized global variable and at least one uninitialized global variable, and compile again. (4) Delphi does not accept library files. You must extract the necessary `.obj` files from the `.lib` file first. For further information, see the article "Using C object files in Delphi" at rvelthuis.de/articles/articles-cobjs.html. If you have problems making this work, then make a DLL and use dynamic linking instead.

11.6 Can I convert an executable file from one system to another?

No. It is not possible to convert executable files between systems because they contain incompatible system calls. It may be possible to find an emulator that can run the executable. For example, the Wine emulator can run Windows executables under Linux if you are lucky.

11.7 Can I convert from 32 bit code to 64 bit code?

No. The instruction codes are not compatible.

11.8 Can I convert a dynamic link library to another system?

No. Objconv does not support the conversion of dynamic link libraries and shared objects.

11.9 Can I build a function library that works in all operating systems?

Yes. It is possible to build a static function library that works in all 32-bit or all 64-bit x86 systems. It is preferably coded in assembly language. See the instructions above.

11.10 Why can't I convert an export library?

The export library contains no function code. It contains only references to a DLL.

11.11 Can I convert a static library to a dynamic library?

Yes. You don't need objconv for this. The linker can do this. You only have to add a simple entry function. The manual for the linker should explain how to do this.

11.12 Can I convert a dynamic library to a static library?

No. If the source code is not available then you will have to disassemble the DLL and identify the function or functions you need. Then re-assemble this code. This is no easy job, but it may be possible in simple cases.

11.13 Can I convert a Windows function library to use it under Linux?

It is possible only in simple cases. See the instructions above for converting compiler-generated code.

11.14 Can I convert a Linux function library to use it under Windows?

It is possible only in simple cases. See the instructions above for converting compiler-generated code.

11.15 I want to know which library contains a particular function

You can make a script that lists the contents of multiple libraries. In Windows, make a file named `listall.bat` containing this line:

```
for %%x in ( *.lib ) do objconv -d %%x >> libraries.txt
```

Make sure `objconv.exe` is in the path, and run `listall.bat` in the directory containing the `.lib` files.

For Linux, make a script file, for example named `listall.sh`, containing the lines below, and make it executable:

```
#!/bin/bash
for x in `ls *.a` ; do ./objconv -d $x >> libraries.txt ; done
```

These scripts will make a text file listing the functions of each library. Use any text editor or search tool to search through the `libraries.txt` file for the function name you are looking for.

11.16 How do I know if my Linux function uses the red zone?

64-bit Unix systems allow functions to use the red zone. There is no red zone in 32-bit systems. You can avoid the red zone by compiling with option `-mno-red-zone`. The compiler doesn't always use the red zone, even without this option. The only way to find out if an object file uses the red zone is to inspect a disassembly. This can be quite difficult.

A converted Linux function that uses the red zone is likely to work in Windows. But there is a theoretical possibility that it will fail with an extremely low frequency. The failure will not be reproducible and thus difficult to track.

11.17 How do I know if my Linux function has position-independent code

Objconv will issue an error message if you try to convert an object file that contains addressing modes that are incompatible with the target system. You will see no error message if objconv is able to work around the problem.

11.18 I have problems porting my Windows application to Linux because the Gnu compiler has a more strict syntax. Can I convert the compiled Windows code instead?

While you are trying to solve a small problem you are creating a much bigger problem instead. There are so many compatibility problems when converting compiler-generated code that this method is unlikely to work. Try to use a compiler that supports both operating systems, such as Gnu or Intel.

11.19 Is it possible to extract one or more functions from a binary file or program?

It is possible to extract modules from a library file (`*.lib`, `*.a`), but it is not possible to automatically extract a function from an object file, executable file or dynamic link library. The file may contain spaghetti code that makes it impossible for the objconv program to tell where each function begins and ends. You may look at a disassembly to search for the function you need. If it is clear where the function begins and ends, and if the function is independent of other functions and data, then it is possible to isolate this function as assembly code and assemble it again. You have to be an assembly expert to do this.

11.20 Is it possible to convert mangled function names?

It is very tedious to do this manually. As yet there is no tool available for converting mangled names automatically. The Microsoft mangled names contain more information than the Gnu mangled names do, so it would be preferable to convert from Windows to Linux rather than vice versa. Se my manual 5: "Calling conventions for different C++ compilers and operating systems".

11.21 Is it possible to convert function calling conventions automatically?

No conversion is needed when converting between different 32-bit systems, except for class member functions using the Microsoft `__thiscall` convention and in rare cases differences in stack alignment. A conversion is needed when converting 64-bit object files because Windows and Linux systems use different calling conventions in 64-bit mode. The standard call stubs supplied with objconv can take care of the most common cases (see page 20).

It might be possible, at least in principle, to construct a tool that makes a specific call stub automatically based on the information of function parameter types contained in mangled function names. This would not work, however, for parameters of composite type because the mangled function names do not contain enough information to predict how a class object parameter is transferred. I am not going to build such a tool.

11.22 Does the disassembler have an interactive feature?

No. The current version of objconv has no feature for manually telling the disassembler what is code and what is data, etc. The disassembler does this automatically except in the most difficult cases.

11.23 Is it possible to disassemble an executable file to modify it and then assemble it again?

The disassembly of an executable program file is unlikely to contain enough information for reconstructing a fully working executable. It may be possible to do this on a DLL in simple cases, but this would be quite difficult.

11.24 Is it possible to disassemble an object file and fix all compatibility problems manually?

If you are an expert, yes. Many compatibility problems can be fixed manually. But this is hard work and there are many pitfalls. This is not for the faint-hearted!

11.25 Is it possible to reconstruct C++ code from a disassembly?

Reconstructing the logic behind a code from the disassembly is a lot of detective work, but it is possible with very small files. The disassembly of a program file typically contains hundreds of thousands of code lines. Interpreting so much code is simply an unmanageable job.

11.26 Why do I get error messages in the disassembly file?

Most disassembly errors occur because the compiler has placed data in the code segment and the disassembler attempts to interpret these data as code. The disassembler does its best to distinguish between code and data, but it is not always successful at this.

Another common cause of errors is misplaced labels caused by cross-references with calculated addresses that the disassembler has interpreted incorrectly.

The disassembler will sometimes show the same binary data both as code and as data if it is in doubt what it is.

Data in the code segment should be avoided because this leads to inefficient caching and code prefetching. Unfortunately, some compilers are still putting jump tables etc. in the code segment. Older compilers do this a lot.

11.27 How does the disassembler distinguish between code and data?

The first assumption is that code segments contain code and data segments contain data. Unfortunately, some compilers put jump tables and other data into the code segment, even though this gives inferior performance. The disassembler follows all cross-references in the code in order to detect the type of each reference target. If something in the code segment is referenced as data it will be labeled as data. If an unreferenced sequence in a code segment begins with several zeroes it will be interpreted as data. If a sequence in a code segment cannot be disassembled without errors it will be interpreted as data or as dubious.

The disassembler will analyze the code preceding any indirect jump or indirect call in an attempt to identify various types of jump tables and virtual tables. Absolute, self-relative and image-relative jump tables are distinguished based on the address-calculating code that precedes an indirect jump.

The distinction between code and data can fail in the following cases:

- If the disassembler has not found any reference to a data object in a code segment
- If a data segment contains code
- If there is self-modifying code
- If a piece of code or data is referenced through a calculated pointer, the disassembler may not be able to completely follow the calculation. This may result in a misplaced label where the disassembler wrongly assumes that the pointer points to. A misplaced label will lead to misinterpretation of whatever follows the label. This is the most common reason for code being interpreted out of phase.

11.28 Can I disassemble byte code?

Objconv cannot convert or disassemble the byte code that is used for .net or Java. There may be other tools available for this.

11.29 Can I assemble the output of the disassembler?

Yes. The output is intended to be fully compatible with the MASM, TASM, NASM, YASM and Gas assemblers. Select the appropriate syntax dialect on the command line. See page 13 for possible compatibility problems.

11.30 Why does the disassembler not support AT&T syntax?

The AT&T syntax is used for compiler-generated code in the Gnu assembler. This syntax is difficult to use and confusing because the operands are written in an order that differs from the code manuals from Intel and AMD. This becomes increasingly difficult with the newest instructions that can have up to five operands.

Most versions of the Gnu assembler support the standard Intel syntax, which is easier to use. The Gnu assembler on Macintosh systems may not support Intel syntax. Use another assembler instead.

It is important when using the Gnu/Intel syntax to put the directive `.intel_syntax noprefix` in the beginning of the code. In case of inline assembly for the Gnu compiler, you must end with `.att_syntax prefix` in order to enable the compiler-generated AT&T code that may follow.

11.31 How can I convert assembly syntax?

You can convert an assembly file from one syntax to another by assembling it to an object file with the appropriate assembler and then disassembling the object file to the desired syntax with objconv. The names of local labels and other details may be lost in the process.

The Intel C++ compiler for Linux supports inline assembly with both AT&T and MASM syntax. This may be used for converting MASM or Intel-style instructions to AT&T syntax, but directives etc. are not supported.

An alternative to converting assembly syntax is to assemble with the appropriate assembler and then converting the resultant object file to the desired file format using objconv.

11.32 Why does my disassembly take so long time?

The handling of symbol tables etc. in objconv is not optimized for very large files. Converting or disassembling files of megabyte size can sometimes take a long time. The handling of small to medium size files goes very fast.

11.33 How can I save the output of the dump screen to a file?

```
objconv -dhs myfile > outputfile.txt
```

11.34 Can you help me with my problems?

No. I am not doing programming work for others. Sorry.

11.35 Are there any alternatives to objconv?

There are certain alternative tools that can convert and manipulate object files.

The Gnu `objcopy` utility can convert between various object file formats. The `objcopy` utility can be recompiled to support the file formats you need.

Intel's C++ compiler can compile the same source code on both Windows, Linux, BSD and Mac OS X platforms (www.intel.com). There are various versions of the Gnu C++ compiler for all platforms as well, although the Windows version is currently not fully up to date.

The NASM, YASM and JWASM assemblers can assemble the same source code for different object file formats.

The Microsoft linker and library manager can convert from 32-bit OMF to COFF. The `Editbin` tool that comes with Microsoft compilers can convert from 32-bit OMF to COFF and modify COFF files.

The Digital Mars compiler includes a tool named `COFF2OMF` for converting 32-bit COFF files to OMF.

There are several other disassemblers available of variable quality, some free and some commercial.

The `tdump` utility that comes with Borland compilers is useful for dumping COFF and OMF files, including executable files.

`debug.exe`. Comes with most versions of Windows. Can disassemble, debug and modify 16-bit executables.

12 Warning and error messages

All possible warning and error messages are listed in the source code in the file `error.cpp`. Below are listed some of the messages that require further explanation.

1050	"Position dependent references will not work in .so file". Shared objects in Linux, BSD and Mac systems require position-independent code. The code you are converting is position-dependent. It will work if statically linked into an executable, but not in a shared object.
1051	Weak public not supported in target file type, symbol xxx. Objconv has changed a public symbol to non-weak. If this symbol clashes with other symbols having the same name then change its name or hide it.
1061	Symbol xxx has lazy binding. Objconv attempts to change the external symbol to non-lazy binding. This usually works when converting from Mac32.
1054	"Cannot find import table". This warning occurs when disassembling an executable file and the disassembler lacks support for recognizing the import table in the file type in question. Some imported symbol names may be missing or wrong in the disassembly output.
1300	"File contains 32-bit absolute address". This can occur when converting from 64-bit ELF to Mach-O and the file

contains 32-bit addresses. Linux and BSD allow 32-bit absolute addresses in 64-bit files because they keep all addresses below 2^{31} (the limit of a *signed* 32-bit addresses). The OS X Darwin system does not allow this because all addresses are usually above 2^{32}. It is possible to work around the problem by specifying an image base less than 2^{31} to the linker in order to keep addresses within the 32-bit address space. Objconv must know the value of the image base so that it can convert the not-allowed 32-bit absolute address to a 32-bit image-relative address. You must specify the same image base to objconv and to the linker. Objconv will use the value 400000 (hexadecimal) if not specified. The following example shows how to build the executable:

```
objconv -fmac64 -imagebase=400000 f1elf.o f1mac.o
g++ -m64 -image_base 400000 -pagezero_size 1000 main.cpp f1mac.o
```

pagezero_size must be ≤ image_base. All numbers are hexadecimal.

1301 "Image-relative address converted to absolute".
This can occur when converting from 64-bit COFF to ELF and the file contains addresses relative to the image base. This addressing mode is not supported in ELF. Objconv can convert the image-relative address to an absolute address if it knows the value of the image base. You can specify a desired image base to objconv and specify the same image base to the linker. For example:

```
objconv -felf64 -imagebase=400000 file1.obj file1.o
g++ -m64 --image-base 400000 main.cpp file1.o
```

If your version of the Gnu linker doesn't accept the `--image-base` command then you must find out which image base it uses and set this value in the objconv command line. The image base must be less than 0x80000000 for the conversion to work. The addresses are all hexadecimal.

2042 "Relocation to global offset table found. Cannot convert position-independent code".
The object file contains position-independent code using a global offset table (GOT). Objconv does not support the conversion of this type of code.

12.1 Linker errors:

_atexit or __cxa_atexit unresolved external

The program is registering a destructor to be called after main() has finished. This is not compatible among systems. You may fix the linker error by making a dummy function with this name that does nothing, but the destructor will not be called.

___cxa_guard_acquire __cxa_guard_release unresolved externals

These functions are locks used by the Gnu compiler to make the initialization of local static objects thread-safe. You may Insert dummy functions for these:

```
extern "C" void __cxa_guard_acquire(){};
extern "C" void __cxa_guard_release(){};
```

__gxx_personality_v0 unresolved external

Make sure that objconv strips exception information (option -xs).
If you get this error on a Unix target system then make sure you compile with `g++`, not `gcc`. If you get this error on a Windows target system then make a dummy variable with this name:

```
extern "C" int _gxx_personality_v0 = 0;
```

kernel32.lib missing

This library is needed by Windows command line compilers. You need to download Microsoft Software Development Kit to get this library. There are two versions of `kernel32.lib`. The 32-bit version is in the `Lib` directory, the 64-bit version with the same name is in `Lib\x64`.

The Mac linker says that the table of contents is out of date.

Some versions of the Mac linker (`ld`) makes an error message if the date stamp of a `.a` file has been changed. You can fix the problem by running ranlib on the `.a` file.

13 Source code

The source code can be used for building the objconv executable for a particular platform and for modifying the program. The code is in C++ language and can be compiled with almost any modern C++ compiler that supports 64-bit integers on any platform with little-endian memory organization. The code has been tested with Microsoft, Intel and Gnu compilers. The code cannot run on platforms with big-endian memory organization, such as the PowerPC-based Mac.

You don't need to read the rest of this chapter unless you want to modify the source code of objconv.

13.1 Explanation of the objconv source code

The source code is intended to be compatible with all C++ compilers. Any modified code should preferably be tested on more than one compiler, including the Gnu compiler which has the strictest syntax checking.

Unfortunately, the C++ syntax has no standardized way of defining integers with a specific number of bits. Therefore, it is essential that you use the type definitions in `maindef.h` for defining integers with a specific size, e.g. `int32` for a 32-bit signed integer, and `uint32` for an unsigned 32-bit integer.

All dynamic data allocation must use the container classes declared in `containers.h` in order to prevent memory leaks. The following container classes are available:

`CMemoryBuffer` is useful for containing binary data of mixed type. You can append a data object `x` of any type to an instance `A` of `CMemoryBuffer` with `A.Push(&x,sizeof(x))`. You can append a zero-terminated ASCII string `s` with `A.PushString(s)`. You can read a data object `x` of type `mytype` stored in `A` at offset `os` with `x = A.Get<mytype>(os);` or `x = *(mytype*)(A.Buf() + os);` The former method does not work with old versions of the Gnu compiler if `A` is an instance of a template class derived from `CMemoryBuffer`, such as `CELF<>`. Use the type casting method in `CELF` and its descendants.

Note that it is dangerous to make a pointer to an object stored in a container because the internal buffer in the container class instance can be re-allocated when new data are added to the buffer. In some cases, the source code does use the unsafe technique of storing pointers to such data, but only when there is certainty that nothing is added to the container after the pointer has been assigned.

The container class `CFileBuffer` is derived from `CMemoryBuffer`. It adds methods for reading and writing files and for detecting the type of a file.

`CTextFileBuffer`, derived from `CFileBuffer`, is used for ASCII files.

The overloaded operators `>>` and `<<` are used for transferring ownership of a memory buffer from one container to another. It works with all descendants of `CFileBuffer`.

The template classes `CArrayBuf<RecordType>` and `CSList<RecordType>` are used for dynamic arrays where all members have the same type `RecordType`. Instances of these classes can be used as simple arrays with the index operator `[]`. `CArrayBuf` allows `RecordType` to have constructors and destructor, `CSList` does not. A dynamic array of type `CArrayBuf` has a size which cannot be changed after it has been set. A dynamic array of type `CSList` can be appended or resized at any time.

`CSList` is useful for sorted lists. `A.PushSort(x)` will insert object `x` in the list `A` in the right position so that the list is kept sorted at all times. `A.PushUnique(x)` does the same, but avoids duplicates. The sort criterion is determined by defining the operator `<` for `RecordType`.

All conversions of data files are done by a number of converter classes, which are all descendants of `CFileBuffer`. A file buffer can convert the data it contains by creating an object of the appropriate converter class, transferring ownership of its data buffer to the converter class object, letting the converter class do the conversion, and then taking back ownership of the converted data buffer, as shown in this example:

```
void CConverter::OMF2COF() {
   // Convert OMF to COFF file
   COMF2COF conv;                        // Make object for conversion
   *this >> conv;                        // Give it my buffer
   conv.ParseFile();                     // Parse file buffer
   if (err.Number()) return;             // Return if error
   conv.Convert();                       // Convert
   *this << conv;                        // Take back converted buffer
}
```

The operators `>>` and `<<` can transfer ownership of the contained data buffer because the classes `CConverter` and `COMF2COF` are both descendants of `CFileBuffer`.

The converter class `CELF` and its descendants are template classes with all the data structures of 32-bit or 64-bit ELF files as template parameters. This is because of the considerable difference between the data structures in 32-bit and 64-bit ELF files. The templates are instantiated explicitly in the bottom of `elf.cpp`.

The reading and interpretation of command line parameters is done by the class `CCommandLineInterpreter`, which has a single instance `cmd`. `cmd` is a global object so that it can be accessed from all parts of the program without being passed as a parameter.

Another global object is the error handler `err`, which is an instance of the class `CErrorReporter`. All error reporting is done with `err.submit(ErrorNumber)`. Exceptions are not used, for reasons of performance.

The Gnu compiler version 4 has a problem with inheritance from template classes because of an overly strict interpretation of the so-called two phase lookup rule. This problem is circumvented by putting `this->` in front of every access to members of an ancestor class in a class derived from a template class. For example, to access `CELF<>::NSections` from `CELF2COF<>` (which is derived from `CELF<>`), you have to write `this->NSections`. It is recommended to test that the code can be compiled with the Gnu compiler in order to catch these problems.

13.2 How to add support for new file formats

Define an id constant `FILETYPE_NEWTYPE` in `maindef.h` to identify the new file type. Add functionality in `CFileBuffer::GetFileType()` in `containers.cpp` for detecting this file type and its word size (16, 32 or 64 bits). Add a name for this file type to `FileFormatNames[]` in `containers.cpp`.

Define a class `CNewType` derived from `CFileBuffer` with member functions for parsing and dumping files of this type. The class declaration goes into `containers.h`. The definition goes into a new `.cpp` file named after the new type. Define converter classes for converting to and from the COFF or ELF type analogously to the existing converter classes in `converters.h`. Each converter class is derived from the class for the file type you convert from. Add member functions to `CConverter` for each converter class. Add case statements in `CConverter::Go()` in `main.cpp` for each possible conversion. A conversion may go through multiple steps if there is no converter class for direct conversion between the two types. You may also define a converter class for converting from NewType to itself in order to make it possible to modify symbol names in a file of type NewType without converting to one of the base types COFF or ELF and back again.

If the new file type contains x86 or x86-64 code then you may add a converter class for disassembling the new type. See below for the interface to the disassembler.

Note that the different object file formats differ in the way self-relative references are defined in relocation records. ELF and 32-bit Mach-O files define self-relative references relative to the beginning of the relocation source field. COFF and OMF files define self-relative references relative to the end of the instruction needing the reference, as the x86 processors do. The difference between the two methods is equal to the length of the source field plus the length of any immediate operand in the instruction. 64-bit Mach-O files use a mixture of these two methods.

Objconv does not support file types with big endian memory organization.

13.3 How to add features to the disassembler

Only file types based on the x86 instruction set and its many extensions can be handled by the disassembler in objconv.

To add support for disassembling a new file type, you first have to make a converter class, as explained above. The converter class creates an instance of `CDisassembler` and uses the following member functions of `CDisassembler`: Use `CDisassembler::Init` for defining file type and possibly image base. Use `CDisassembler::AddSection` for defining each segment or section. Sections are numbered sequentially, starting at 1. Use `CDisassembler::AddSymbol` for defining local, public and external symbols. These can be numbered in random order, but numbers must be positive and limited. Use `CDisassembler::AddRelocation` for defining all cross-references and relocatable addresses. These can refer to symbol numbers. Use `CDisassembler::Go` to do the disassembly after all sections, symbols and relocations have been defined. Finally, take ownership of the disassembly file `CDisassembler::OutFile`.

You can add support for new instruction codes by adding entries to the opcode tables in `opcodes.cpp`. New Intel opcodes are likely to be 3-byte opcodes beginning with 0F 38 through 0F 3B. These are defined in tables `OpcodeMap2` through `OpcodeMap5`. New AMD opcodes are likely to begin with 0F 24, 0F 25, 0F 7A or 0F 7B defined in tables `OpcodeMap66` through `OpcodeMap69`.

The meaning of each field in the opcode table records is defined in the beginning of `disasm.h`.

Modifications to the functionality of the disassembler go into `disasm1.cpp`. Modifications to the way the disassembly output looks or support for alternative assembly syntaxes go into `disasm2.cpp`.

13.4 File list

Files in objconv.zip	
instructions.pdf	This file
objconv.exe	Executable for Windows
source.zip	Complete source code
extras.zip	Call stubs etc.
Files in source.zip	
build.sh	Script for building objconv for Linux, BSD and Mac systems
objconv.vcproj	Project file for Microsoft compiler
cmdline.cpp	Defines class CCommandLineInterpreter for reading command line
cof2asm.cpp	Defines class CCOF2ASM for disassembling COFF files
cof2cof.cpp	Defines class CCOF2COF for modifying COFF files
cof2elf.cpp	Defines class CCOF2ELF for converting from COFF to ELF
cof2omf.cpp	Defines class CCOF2OMF for converting from COFF to OMF
coff.cpp	Defines class CCOFF for parsing and dumping COFF files
containers.cpp	Container classes CMemoryBuffer, CFileBuffer, CTextFileBuffer
disasm1.cpp	Defines part of class CDisassembler for disassembling
disasm2.cpp	Defines part of class CDisassembler for disassembling
elf.cpp	Template class CELF for dumping and parsing ELF files
elf2asm.cpp	Template class CELF2ASM for disassembling ELF files
elf2cof.cpp	Template class CELF2COF for converting from ELF to COFF
elf2elf.cpp	Template class CELF2ELF for modifying ELF files
elf2mac.cpp	Template class CELF2MAC for converting from ELF to Mach-O
error.cpp	Defines class CErrorReporter and error texts
library.cpp	Defines class CLibrary for building and modifying .lib and .a files
mac2asm.cpp	Defines class CMAC2ASM for disassembling Mach-O files
mac2elf.cpp	Defines class CMAC2ELF for converting from Mach-O to ELF
mac2mac.cpp	Defines class CMAC2MAC for modifying Mach-O files
macho.cpp	Defines class CMACHO for parsing and dumping Mach-O files
main.cpp	Classes CMain and CConverter for dispatching command
omf.cpp	Defines class COMF for parsing and dumping OMF files
omf2asm.cpp	Defines class COMF2ASM for disassembling OMF files
omf2cof.cpp	Defines class COMF2COF for converting from OMF to COFF
omfhash.cpp	Defines class COMFHashTable for hash tables in OMF libraries
opcodes.cpp	Tables for complete set of opcodes for disassembler
stdafx.cpp	Needed only for precompiled headers
cmdline.h	Declares class CCommandLineInterpreter and various constants
coff.h	Structures and constants for COFF files
containers.h	Declares container classes and container class templates
converters.h	Declares many converter classes derived from CFileBuffer
disasm.h	Declares several structures and classes used by disassembler
elf.h	Structures and constants for ELF files
error.h	Declares class CErrorReporter for error handling
library.h	Structures and classes for managing .lib and .a files
macho.h	Structures and constants for Mach-O files
maindef.h	Type definitions and other main definitions
omf.h	Structures, classes and constants for OMF files
stdafx.h	Includes all the other .h files

Files in extras.zip	
u2wstub.obj	Call stub for functions converted from 64-bit ELF or Mach-O to COFF
u2wstubvec1.obj	Same, with 1 vector parameter
u2wstubvec2.obj	Same, with 2 vector parameters
w2ustub.o	Call stub for functions converted from 64-bit COFF to ELF or Mach-O
w2ustubvec.o	Same, with 1 - 4 vector parameters
u2wstub.asm	Source code for u2wstub.obj
u2wstubvec1.asm	Source code for u2wstub.obj
u2wstubvec2.asm	Source code for u2wstub.obj
w2ustub.asm	Source code for w2ustub.o
w2ustubvec.asm	Source code for w2ustub.o

13.5 Class list

The most important container classes and converter classes in the objconv source code are listed below.

Container classes	
CMemoryBuffer	**Declared in:** containers.h **Defined in:** containers.cpp **Inherit from:** none **Description:** This is the base container class that all file classes, converter classes and all classes containing data of mixed types are derived from. The size can grow as new data are added.
CFileBuffer	**Declared in:** containers.h **Defined in:** containers.cpp **Inherit from:** CMemoryBuffer **Description:** This is the container class that all converter classes and other file handling classes are derived from. It adds methods for reading and writing files and for detecting the input file type.
CTextFileBuffer	**Declared in:** containers.h **Defined in:** containers.cpp **Inherit from:** CFileBuffer **Description:** Container class for reading and writing ASCII text files.
CArrayBuf<>	**Declared in:** containers.h **Defined in:** containers.h **Inherit from:** none **Description:** Container class template for arrays where all records have the same type. The record type is defined as a template parameter. The size cannot be modified after it has been set. The record type can have constructors and destructor.
CSList<>	**Declared in:** containers.h **Defined in:** containers.h **Inherit from:** CMemoryBuffer **Description:** Container class template for arrays where all records have the same type. The record type is defined as a template parameter. The size can grow as new records are added. The list can be sorted. The record type can not have constructors or destructor.

Classes for converting files, etc.	
CMain	**Declared in:** converters.h **Defined in:** main.cpp **Inherit from:** CFileBuffer **Description:** Dispatching input file to CConverter or CLibrary

CConverter	**Declared in:** converters.h **Defined in:** main.cpp **Inherit from:** CFileBuffer **Description:** Dispatching input file to any of the converter classes
CLibrary	**Declared in:** library.h **Defined in:** library.cpp **Inherit from:** CFileBuffer **Description:** Reading and building library files of any type
COMFHashTable	**Declared in:** library.h **Defined in:** omfhash.cpp **Inherit from:** none **Description:** Reading and building hash table for OMF libraries
CCOF	**Declared in:** converters.h **Defined in:** coff.cpp **Inherit from:** CFileBuffer **Description:** Parsing and dumping of COFF and PE files
CCOF2ELF	**Declared in:** converters.h **Defined in:** cof2elf.cpp **Inherit from:** CCOFF **Description:** Conversion from COFF to ELF
CCOF2OMF	**Declared in:** converters.h **Defined in:** cof2omf.cpp **Inherit from:** CCOFF **Description:** Conversion from COFF to OMF
CCOF2ASM	**Declared in:** converters.h **Defined in:** cof2asm.cpp **Inherit from:** CCOFF **Description:** Disassembly of COFF and PE files
CCOF2COF	**Declared in:** converters.h **Defined in:** cof2cof.cpp **Inherit from:** CCOFF **Description:** Modification of COFF files
COMF	**Declared in:** converters.h **Defined in:** omf.cpp **Inherit from:** CFileBuffer **Description:** Parsing and dumping of OMF files
COMF2COF	**Declared in:** converters.h **Defined in:** omf2cof.cpp **Inherit from:** COMF **Description:** Conversion from OMF to COFF
COMF2ASM	**Declared in:** converters.h **Defined in:** omf2asm.cpp **Inherit from:** COMF **Description:** Disassembly of OMF files
CELF<>	**Declared in:** converters.h **Defined in:** elf.cpp **Inherit from:** CFileBuffer **Description:** Parsing and dumping of ELF files. The 32-bit or 64-bit ELF structures are defined as template parameters.
CELF2COF<>	**Declared in:** converters.h **Defined in:** elf2cof.cpp **Inherit from:** CELF<> **Description:** Conversion from ELF to COFF. The 32-bit or 64-bit ELF structures are defined as template parameters.
CELF2MAC<>	**Declared in:** converters.h **Defined in:** elf2mac.cpp **Inherit from:** CELF<>

	Description: Conversion from ELF to Mach-O. The 32-bit or 64-bit ELF and MAC structures are defined as template parameters.
CELF2ASM<>	**Declared in:** converters.h **Defined in:** elf2asm.cpp **Inherit from:** CELF<> **Description:** Disassembly of ELF files. The 32-bit or 64-bit ELF structures are defined as template parameters.
CELF2ELF<>	**Declared in:** converters.h **Defined in:** elf2elf.cpp **Inherit from:** CELF<> **Description:** Modifications of ELF files. The 32-bit or 64-bit ELF structures are defined as template parameters.
CMACHO<>	**Declared in:** converters.h **Defined in:** macho.cpp **Inherit from:** CFileBuffer **Description:** Parsing and dumping of Mach-O files. The 32-bit or 64-bit Mach-O structures are defined as template parameters.
CMACUNIV	**Declared in:** converters.h **Defined in:** macho.cpp **Inherit from:** CFileBuffer **Description:** Parsing Mac universal binary files
CMAC2ASM<>	**Declared in:** converters.h **Defined in:** mac2asm.cpp **Inherit from:** CMACHO **Description:** Disassembly of Mach-O files
CMAC2MAC<>	**Declared in:** converters.h **Defined in:** mac2mac.cpp **Inherit from:** CMACHO **Description:** Modifications of Mach-O files and sorting the symbol table. The structures are defined as template parameters
CMAC2ELF<>	**Declared in:** converters.h **Defined in:** mac2elf.cpp **Inherit from:** CMACHO **Description:** Conversion from Mach-O to ELF. The 32-bit or 64-bit MAC and ELF structures are defined as template parameters
CDisassembler	**Declared in:** disasm.h **Defined in:** disasm1.cpp, disasm2.cpp, opcodes.cpp **Inherit from:** none **Description:** Disassembling code. Called from CCOF2ASM, COMF2ASM, CELF2ASM, CMAC2ASM
CSymbolTable	**Declared in:** disasm.h **Defined in:** disasm1.cpp **Inherit from:** none **Description:** Manage symbol table during disassembly.
CErrorReporter	**Declared in:** error.h **Defined in:** error.cpp **Inherit from:** none **Description:** Printing warnings and errors
CCommandLineInterpreter	**Declared in:** cmdline.h **Defined in:** cmdline.cpp **Inherit from:** none **Description:** Interpretation of command line parameters
CResponseFileBuffer	**Declared in:** converters.h **Defined in:** cmdline.cpp **Inherit from:** CFileBuffer **Description:** Contains response file from command line

14 Legal notice

Objconv is an open source program published under the conditions of the GNU General Public License, as defined in www.gnu.org/licenses/. The program is provided without any warranty or support.

It may in some cases be illegal to modify, convert or disassemble copyright protected software files without permission from the copyright owner. It is an open question whether it is legal to modify or convert a copyright protected function library and use it for other purposes than presupposed in the license conditions. It is recommended to ask the vendor for permission before developing and publishing any software that is built with the use of a converted copyright protected function library.

Copyright law does not generally permit disassembly of copyright protected software for the purpose of circumventing a copy protection mechanism, for using part of the code in other contexts, or for extracting the algorithms behind the code.

European, Australian and US copyright law does, however, under certain conditions permit reverse engineering of copyright protected software when the purpose is to extract the information necessary for establishing interoperability with other software, and only to the extent necessary for this purpose. However, I am not a legal expert. The user should seek legal advise before deciding whether it is legal to use objconv on copyrighted software for a specific purpose.