Instructions for objconv

A cross-platform utility for converting and modifying object files and function library files

By Agner Fog

Version 0.91 beta.

Contents

1 Introduction

Objconv is a utility for converting and modifying object files (*.obj, *.o) and static linking library files (*.lib, *.a) for x86 platforms, including 32 and 64 bit Windows, Linux, BSD and Intel-based Mac. Objconv can convert object and library files between the COFF format used in Windows systems, the ELF format used in Linux and BSD and the Mach-O format used in Intel-based Macintosh computers. Support for OMF format is not included in the beta version (0.91), but may be added later. The main purpose of objconv is for cross-platform development of function libraries from assembly source code.

Objconv can also be used for renaming or modifying symbols in object and library files, for dumping files, and as a cross-platform library manager.

A library in this context means a collection of object files. This is called a static linking library file (*.lib) in Windows terminology or an archive (*.a) 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 dynamic link libraries (*.dll), shared object files (*.so) or executable files, but it may be able to dump such files.

Objconv is an open source program under the GNU General Public License, as defined in www.gnu.org/copyleft/gpl.html.

The following table summarizes the type of conversions that objconv can do:

File format	Word size	Extension	Operating system	Convert from	Convert to	Modify	Dump	Disas-semble
COFF	32	.obj, .lib	Windows	x	x	x	x	-
COFF	64	.obj, .lib	Windows	x	x	x	x	-
OMF	16	.obj, .lib	Windows	-	-	-	-	-
OMF	32	.obj, .lib	Windows	-	-	-	-	-
ELF	32	.o, .a	Linux, BSD	x	x	x	x	-
ELF	64	.o, .a	Linux, BSD	x	x	x	x	-
Mach-O	32	.o, .a	Mac OS X	-	x	-	x	-

COFF format is also called PE. Support for OMF and Mach-O formats and disassembly may be added in future versions.

2 Command line syntax

Objconv is executed from a command line or 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`	Desired output format. `XXX` = `COFF`, `ELF` or `MAC`. `PE` is accepted as synonym for `COFF`. The word size, 32 or 64, may be appended to the name, e.g. `ELF64`. The output format must be specified, except for the dump command `-fd`, which has no output.
`-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).
`-ds`	Strip debug information from file. (Default when converting to different format).
`-dp`	Preserve debug information, even if it is incompatible with the target system.
`-xs`	Strip exception handling information and other incompatible info. (Default when converting to 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 nonstandard 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.
`-ar:N1:N2`	Give public symbol `N1` an alias name `N2`. The same symbol will be accessible as `N1` as well as `N2`.
`-nw:N1`	Make public symbol `N1` weak. Only possible for ELF 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 `:`.
`-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.
`-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.
`-h`	Help. Print list of options.
`@RFILE`	Read additional options from response file `RFILE`.

3 Converting file format

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 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.

A library is converted in the same way:

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

The output file will always have the same word size as the input file. It is not possible to change the word size.

You may use the `-nu` option to add or remove leading underscores on symbol names.

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. You may get an unresolved reference named `__gxx_personality_v0` when converting from Gnu systems to another format if the exception handling information is not removed.

The input file should preferably be generated from assembly code with careful observation of the calling conventions of the target system. Compiler-generated code should not be converted but recompiled on the target platform. If the source code is not available then it may be necessary to disassemble the object file, fix any incompatibilities in the assembly code, and then assemble again. Linux, BSD and Mac systems are very similar and may be compatible with each other without recompiling.

The reasons why conversion of compiler-generated code to a different format may not work can be summarized as follows:

Reasons why conversion of compiler-generated code may fail	
Calling conventions in 64-bit mode	The calling conventions in 64-bit Windows and 64-bit Linux are very different. Functions with integer parameters will not work. Windows functions may use a shadow space not available in Linux. Linux functions may use a red zone not available in Windows.
Calling conventions in 32-bit mode	Most compilers have the same calling conventions in 32-bit mode, except for class member functions in Microsoft-compatible compilers. Use the keyword `__cdecl` on member function declarations on Microsoft-compatible compilers to force a compatible calling convention, or use `friend` functions instead of member functions.
Register usage conventions in 64-bit mode	Linux functions may modify registers `RSI`, `RDI` and `XMM6` - `XMM15`, which must be preserved by Windows functions.
Register usage conventions in 32-bit mode	The register usage conventions are the same in all 32-bit systems except for Watcom compilers.
Mangling of function names	Different compilers use different name mangling schemes. Use `extern "C"` on all function declarations 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`.
Leading underscores on function names	Use the `-nu` option on `objconv` to add or remove leading underscores when converting 32-bit files.
Initialization and termination code	Initialization and termination code is used for calling the constructors and destructors of global objects and for initializing function libraries. This code is not compatible between different systems.
Exception handling and stack unwinding information	This information is not compatible between different systems. Do not rely on structured exception handling.
Virtual tables and runtime type identification	Do not use virtual member functions or runtime type identification.
Communal functions	`Objconv` does not include a feature for converting communal

and data	(coalesced) data. Do not use function-level linking (`/Gy`) on Microsoft compilers or `-ffunction-sections` on Gnu compilers.
Compiler-specific library calls	Most compilers can generate calls to library functions that are specific to that particular compiler. It may be necessary to convert the library function as well or make a replacement for this function.
Calls to operating system	Operating system calls are not compatible among systems.
Position-independent code	The Mac OS X operating system requires position-independent code if you are making shared objects. Other systems may not be able to make position-independent code. Use static linking instead of dynamic linking when using converted object files on Mac systems. See `macpic.asm` for instructions on how to make position-independent code in MASM.

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).

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

4 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. Use the dump option to see the mangled symbol names.

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

```
objconv -fcoff -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 public symbol more than one name. This can be useful for the purpose of supporting multiple name mangling schemes with the same object or library file. To give the function named `function1` the alias `function2`:

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

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

It is possible to make a public symbol weak in ELF files. A weak symbol has low 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 file format allows this feature. To make public symbol `function1` weak in ELF file `file1.o`:

```
objconv -felf -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 used by any other modules. To hide symbol `DontUseMe` in COFF file `file1.obj`:

```
objconv -fcoff -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.

5 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. `Objconv` will always modify the member names, if necessary, to meet the following restrictions:

- Any path is removed from the member name.
- The member name (including extension) cannot be longer than 15 characters. This is for the sake of compatibility between different systems that differ in the way longer names are stored. The name is truncated if necessary.
- The member names must be different. Names that become identical after truncation will be modified to make them different.
- The member name must end in `.obj` for COFF and OMF files, or `.o` for ELF and Mach-O files. The extension is changed or added if necessary.

The libraries contain a symbol index in order to make it easier for linkers to find a particular symbol. `Objconv` will always remake the symbol index whenever a library file is modified.

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

Rebuilding a library

To remove any path from member names and rebuild the symbol table in library `mylib.lib`:

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

or alternatively:

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

The `-lib` option 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 write:

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

Extracting members from a library

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

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

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

```
objconv -felf -lx mylib.a
```

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

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

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

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

Dumping library contents

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

```
objconv -fd 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 and the length has been limited to 15 characters.

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

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

6 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, to convert errors to warnings and to convert warnings to errors.

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.

7 Using a response file

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.

8 Alternative tools

There are certain alternative tools that can convert object file formats. The Microsoft linker and library manager can convert from 32-bit OMF to COFF. Some compilers that use the OMF format include a utility for converting 32-bit COFF to OMF. The Gnu `objcopy` utility can convert between various object file formats. The `objcopy` utility can be recompiled to support the file formats you need.

9 Source code

The source code can be used for rebuilding the executable for a particular platform and for modifying the objconv program. The code is in C++ language and can be compiled with almost any modern C++ compiler that supports 64-bit integers.

Project files for Microsoft and Gnu compilers are included. To build objconv with another compiler, just make a project that includes all the `.cpp` files and compile for console mode.

10 File list

objconv.exe	Executable for Windows
instructions.pdf	This file