Object file

Condensed Version

From Wikipedia, the free encyclopedia.

In computer science, **object file** or **object code** is an intermediate representation of code generated by a compiler after it processes a source code file. Object files contain compact, pre-parsed code, often called binaries, that can be linked with other object files to generate a final executable or code library. An object file is mostly machine code (data directly understandable by a computer's CPU).

An **object file format** is a computer file format used for the storage of object code and related data typically produced by a compiler or Assembler.

An object file contains not only the object code, but also relocation information that the linker uses to assemble multiple object files into an executable or library, program symbols (names of variables and functions), and debugging information.

There are many different object file formats; originally each type of computer had its own unique format, but with the advent of Unix and other portable operating systems, some formats, such as COFF and ELF, have been defined and used on different kinds of systems. It is common for the same file format to be used both as linker input and output, and thus as the library and **executable file format**.

The design and/or choice of an object file format is a key part of overall system design; it affects the performance of the linker and thus programmer turnaround while developing, and if the format is used for executables, the design also affects the time programs take to begin running, and thus the responsiveness for users. Most object file formats are structured as blocks of data all of the same sort; these blocks can be paged in as needed by the virtual memory system, needing no further processing to be ready to use.

The simplest object file format is the DOS COM format, which is simply a file of raw bytes that is always loaded at a fixed location. Other formats are an elaborate array of structures and substructures whose specification runs to many pages.

Debugging information may either be an integral part of the object file format, as in COFF, or a semi-independent format which may be used with several object formats, such as stabs or DWARF. See debugging format.

The GNU project's BFD library provides a common API for the manipulation of object files in a variety of formats.

Types of data supported by typical object file formats:

- BSS ("block started by symbol")
- text segment
- data segment

Notable object file formats:

- COM (DOS)
- EXE (DOS)
- a.out (Unix / Linux)
- COFF (Unix / Linux)
- XCOFF (AIX)
- ECOFF (Mips)
- ELF (Unix / Linux)
- SOM (HP)
- Mach-O (NeXT, Mac OS X)
- Portable Executable (Windows)
- NLM
- OMF
- PEF (Macintosh)
- IEEE-695 (embedded)
- S-records (embedded)
- IBM 360 object format

Reference

- John R. Levine, *Linkers and Loaders* (*http://www.iecc.com/linker/*) (Morgan Kaufmann Pub, 2000)

A.out

From Wikipedia, the free encyclopedia.

*The title of this article is incorrect because of technical limitations. The correct title is **a.out**.*

a.out is the default filename for executable output from many compilers, especially in Unix environments. It stands for "assembler output".

a.out is also an old object file format for executables, now superseded by the ELF and COFF formats.

(These two meanings are easily confused, as e.g. the default object file *format* of a compile/linker might be ELF and used to be a.out in earlier versions, but the default file *name* is still "a.out".)

COFF

From Wikipedia, the free encyclopedia.

The **Common Object File Format** (**COFF**) is an object file format that was introduced in Unix System V Release 3, and was later adopted by Microsoft for Windows NT. It was superseded by the more powerful ELF in System V Release 4, but as of 2005 COFF is still used in Windows as Portable Executable.

The original Unix object file format a.out is a very simple design, and was too limited to effectively handle the additions of SVR3, such as shared libraries. Also, a.out does not define a symbolic debug data format; stabs works by encoding debug info into special symbols in the symbol table.

COFF's main improvement was the introduction of multiple named *sections* in the object file. Different object files could have different numbers and types of sections. In addition, a debug data format was defined.

However, the COFF design soon turned out to be too limited; there was a limit on the maximum number of sections, a limit on the length of section names, and so forth. In addition, the debug info was really only capable of supporting C debugging; for instance, C++ had additional constructs that had no way to be represented, and the debug info was designed to be extensible. IBM solved this in AIX with the XCOFF format, MIPS and others used ECOFF, and GNU tools adopted the workaround of encoding stabs info, which was extensible, into special COFF sections in a technique known as stabs-in-coff.

Executable and Linkable Format

From Wikipedia, the free encyclopedia.

The **Executable and Linkable Format** (**ELF**) is a common standard in computing for executables and object code. First published in the Tool Interface Standard and the System V Application Binary Interface, it was quickly accepted among different vendors of UNIX systems.

Today the ELF format has replaced the proprietary (or sometimes just platform-specific) and less extensible executable formats (primarily COFF) in the Linux, Solaris, Irix, and almost all modern BSD operating systems.

Other object code file formats are a.out and COFF; ELF could be considered a "competitor" to those, although it is generally considered to outperform both of them. [edit]

ELF file layout

Each ELF file is made up of one ELF header, followed by zero or more segments and zero or more sections. The segments contain information that is necessary for runtime execution of the file, while sections contain important data for linking and relocation. Each byte in the entire file is taken by no more than one section at a time, but there can be orphan bytes, which are not covered by a section. In the normal case of a UNIX executable one or more sections are enclosed in one segment. The segments and sections of the file are listed in a program header table and section header table respectively.

On many UNIX systems the command

```
man elf
```

may provide some more details. [edit]

Tools

- `readelf` is a UNIX binary utility that displays information about one or more ELF files. A GPL implementation is provided by GNU Binutils
- `elfdump` is a Solaris command for viewing ELF information in an elf file.

External links

- Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification Version 1.2 (*http://x86.ddj.com/ftp/manuals/tools/elf.pdf*)
- Description of the ELF binary format (*http://www.cs.ucdavis.edu/~haungs/paper/node10.html*)
- Article "LibElf and GElf - A Library to Manipulate ELF Files (*http://developers.sun.com/solaris/articles/elf.html*)" by Neelakanth Nadgir
- free ELF object file access library (*http://www.stud.uni-hannover.de/~michael/software/english.html*)
- manual page (*http://www.dac.neu.edu/cgi-bin/man-cgi?libelf*)
- Elf library routines

HEX File Format

A HEX file is made of HEX records, one record per line. The format for the HEX record is as follows:

1. Start with a colon (:);
2. Then two hex digits give the number of data bytes in the record.

3. Then two four hex digits give the starting address of where to place the data in memory.
4. Then there is a two hex-digit identifying tag. The only two values you care about are 00 for data records and 01 for the terminating record in the file.
5. Then there are N two-digit hex numbers where N is the number of data bytes in the record.
6. Finally there is a two-digit hex checksum which is the lower two digits of the two's-complement of the sum of all the previous two-digit fields (the four-digit address is split into two 2-digit fields).

Here is an example file that shows the HEX format:

```
:1000800082D08322C083C08290F000F0D082D083DF
:030090002200004B
:00000001FF
```

You can get some example code for this from the XSTOOLs source code in http://www.xess.com/FPGA/xstools.zip.

Reference: http://xess.com/faq/M0000181.HTM

The HEX file format has a number of benefits: they are simply to generate, and to interpret; they provide the ability to load bytes into any address, not necessarily in a sequential order; and they are, to some degree, human-readable and manually modifiable. HEX files are by no means perfect; one of the main problem with them is that the HEX file format is not really a standard. For instance, some HEX files have 16-bit addresses (taking 4 hex digits), others 32-bit (which take 8 hex digits) – so you have to ensure that you use the correct interpreter to read a given HEX format. Sometimes the checksum at the end is also implemented in a different way, and this sort of thing tends to discourage the use of HEX files in situations where it is important that the executable be represented in a standard format. Perhaps, if a HEX file had a standard mechanism by which to identifying these variations on the theme, they would become an acceptable standard; but then this would make the format more complex.

BIN Files

The nice thing about binary (BIN) files, are they that they don't conform to any standard. They simply contain raw, unadulterated, plain binary data. A BIN file typically contain a verbatim byte-for-byte representation of what is to be put into memory. When using BIN files, you have to tell the linker which address the binary file starts at, and you must ensure that the binary file is loaded to that address on the target platform before you execute it. It is customary to put the start() function as the first entry in a BIN file, unless the BIN file contains bytes (such as vector table values) which are placed in lower addresses to the start function.

One thing to watch out for, when using BIN files, are gaps between memory segments. Consider, for instance, that you have a 32 byte exception vector starting at address 0x00000000, which goes into SRAM, and then a large gap until address 0x20100000 which contains the application code that goes into SDRAM. If you tell the linker these two addresses, and then simply generate a BIN file from this configuration, you will find that the resultant BIN file is a number of megabytes, i.e. contains a copy of what should go into memory starting at address 0 until the last application address. In this situation, you either need to use use two binary files, loading, for the example the vectors to address 0, and then the application to address 0x20100000; or you need to use a different object file format (such as ELF), or you need to be inventive; such as making the application's start() function assign the exception vectors.

Binary files are generally used in cases where there is either no loader (e.g. the binary file contains the loader program), or for downloading programs directly into RAM without having to process a file format (this is often the case when you are testing or debugging embedded code on the target platform, and are frequently downloading slightly modified application programs).

OBJCOPY *-- part of GNU BinTools*

Section: GNU Development Tools (1)

NAME

objcopy - copy and translate object files

SYNOPSIS (excerpt)

objcopy [**-F** *bfdname*|**--target=***bfdname*]
[**-I** *bfdname*|**--input-target=***bfdname*]
[**-O** *bfdname*|**--output-target=***bfdname*]
[**-S**|**--strip-all**] [**-g**|**--strip-debug**]
[**-N** *symbolname*|**--strip-symbol=***symbolname*]
[**-R** *sectionname*|**--remove-section=***sectionname*]
[**--gap-fill=***val*] [**--pad-to=***address*]
[**--set-start=***val*] [**--adjust-start=***incr*]
[**--change-section-address** *section*{=,+,-}*val*]
[**--set-section-flags** *section=flags*]
[**-v**|**--verbose**]
[**-V**|**--version**]
[**--help**]
infile [*outfile*]

DESCRIPTION

The GNU **objcopy** utility copies the contents of an object file to another. **objcopy** uses the GNU BFD Library to read and write the object files. It can write the destination object file in a format different from that of the source object file. The exact behavior of **objcopy** is controlled by command-line options. Note that **objcopy** should be able to copy a fully linked file between any two formats. However, copying a relocatable object file between any two formats may not work as expected.

objcopy creates temporary files to do its translations and deletes them afterward. **objcopy** uses BFD to do all its translation work; it has access to all the formats described in BFD and thus is able to recognize most formats without being told explicitly.

objcopy can be used to generate S-records by using an output target of **srec** (e.g., use **-O srec**).

objcopy can be used to generate a raw binary file by using an output target of **binary** (e.g., use **-O binary**). When **objcopy** generates a raw binary file, it will essentially produce a memory dump of the contents of the input object file. All symbols and relocation information will be discarded. The memory dump will start at the load address of the lowest section copied into the output file.

When generating an S-record or a raw binary file, it may be helpful to use **-S** to remove sections containing debugging information. In some cases **-R** will be useful to remove sections which contain information that is not needed by the binary file.

Note - **objcopy** is not able to change the endianness of its input files. If the input format has an endianness, (some formats do not), **objcopy** can only copy the inputs into file formats that have the same endianness or which have no endianness (eg **srec**).