COFF –通用对象文件格式（Common Object File Format），是一种很流行的对象文件格式（注意：这里不说它是“目标”文件，是为了和编译器产生的目标文件（\*.o/\*.obj）相区别，因为这种格式不只用于目标文件，库文件、可执行文件也经常是这种格式）。大家可能会经常使用VC吧？它所产生的目标文件（\*.obj）就是这种格式。其它的编译器，如GCC（GNU Compiler Collection）、ICL（Intel C/C++ Compiler）、VectorC，也使用这种格式的目标文件。不仅仅是C/C++，很多其它语言也使用这种格式的对象文件。

目标文件

统一格式的目标文件为混合语言编程带来了极大的方便。

当然，并不是只有这一种对象文件格式。常用格式的还有OMF-对象模型文件（Object Module File）以及ELF-可执行及连接文件格式（Executable and Linking Format）。OMF是一大群IT巨头在n年制定的一种格式，在Windows平台上很常见。大家喜欢的Borland公司使用的目标文件就是这种格式。MS和Intel之前用的也是这种格式。而投异侧，用COFF格式了。ELF格式在非Windows平台上使用得比较多，在Windows平台基本上没见过。

结构

COFF文件的整体结构：

File Header

Optional Header

Section Header 1

......

Section Header n

Section Data

Relocation Directives

Line Numbers

Symbol Table

String Table

如上图：

COFF文件一共有8种数据，自上而下分别为：

1.文件头（File Header）20Bytes

2. 可选头（Optional Header）0bytes

3. 段落头（Section Header）40bytes

4. 段落数据（Section Data）

5.重定位表（Relocation Directives）n\*10Bytes

6. 行号表（Line Numbers）

7.符号表（Symbol Table）m\*18Bytes

8.字符串表（String Table）

其中，除了段落头可以有多个节（因为可以有多个段落）以外，其它的所有类型的节最多只能有一个。

文件头：用来保存COFF文件的基本信息，如文件标识，各个表的位置等等。

可选头：是可选的，可有可无的。在目标文件中，基本上都没有这个头；但在其它的文件中（如：可执行文件）这个段用来保存在文件头中没有描述到的信息。

段落头：是用来描述段落信息的，每个段落都有一个段落头来描述。段落的数目在文件头中会指出。

段落数据：这通常是COFF文件中最大的数据段，每个段落真正的数据就保存在这个位置。

重定位表：这个表通常只存在于目标文件中，它用来描述COFF文件中符号的重定位信息。至于为什么要重定位，请回家看看你的操作系统的书籍。

符号表：这个表用来保存COFF文件中所用到的所有符号的信息，连接多个COFF文件时，这个表帮助我们重定位符号。调试程序时也要用到它。

字符串表：用来保存字符串的。符号表是以记录的形式来描述符号信息的，但它只为符号名称留置了8个字符的空间，早期的小程序还将就能行，现代程序中，一个符号名动不动就数十个字符，8个字符怎么能够？没办法，只好把这些名称存在字符串表中。而符号表中只记录这些字符串的位置。

文件的结构大体上就是这样了。它的设计者还是有点远见的。可扩充性设计得不错，以致于沿用至今。了解了文件的整体结构，让我们来逐个段落分析它。

文件头

文件头，从文件的0偏移处开始，它的结构很简单。用C的结构描述如下：

typedef struct {

unsigned short usMagic; //2魔法数字

unsigned short usNumSec; //2 段落（Section）数

unsigned long ulTime; //4 时间戳

unsigned long ulSymbolOffset; //4 符号表偏移

unsigned long ulNumSymbol; //4 符号数

unsigned short usOptHdrSZ; //2 可选头长度

unsigned short usFlags; //2 文件标记

} FILEHDR;

结构中usMagic成员是一个魔法数字（Magic Number），在I386平台上的COFF文件中它的值为0x014c。如果COFF文件头中魔法数字不为0x014c，那就不用看了，这不是一个I386平台的COFF文件。其实这就是一个平台标识。

第二个成员usNumSec是一个无符号短整型，它用来描述段落的数量。段落头（Section Header）的数目就是它。

ulTime成员是一个时间戳，它用来描述COFF文件的建立时间。当COFF文件为一个可执行文件时，这个时间戳经常用来当做一个加密用的比对标识。

ulSymbolOffset是符号表在文件中的偏移量，这是一个绝对偏移量，要从文件头开始计数。在COFF文件的其它节中，也存在这种偏移量，它们都是绝对偏移量。

ulNumSymbol成员给出了符号表中符号记录的数量。

usOptHdrSZ是可选头的长度，通常它为0。而可选头的类型也是从这个长度得知的，针对不同的长度，我们就要选择不同的处理方式。

usFlag是COFF文件的属性标记，它标识了COFF文件的类型，COFF文件中所保存的数据等等信息。其值如下：

|  |  |  |
| --- | --- | --- |
| 值 | 名称 | 说明 |
| 0x0001 | F\_RELFLG | 无重定位信息标记。这个标记指出COFF文件中没有重定位信息。通常在目标文件中这个标记们为0，在可执行文件中为1。 |
| 0x0002 | F\_EXEC | 可执行标记。这个标记指出 COFF 文件中所有符号已经解析， COFF 文件应该被认为是可执行文件。 |
| 0x0004 | F\_LNNO | 文件中所有行号已经被去掉。 |
| 0x0008 | F\_LSYMS | 文件中的符号信息已经被去掉。 |
| 0x0100 | F\_AR32WR | 些标记指出文件是 32 位的Little-EndianCOFF 文件。 |

注：Little-Endian，是指小字节序或低字节序。它是指数据的排列方式。比如：十六进制的0x1234以Little-Endian方式在内存中的顺序为0x34 0x12。与之相反的是Big-Endian，这种方式下，在内存中的顺序是0x12 0x34。

可选头

可选头接在文件头的后面，也就是从COFF文件的0x0014偏移处开始。长度可以为0。不同长度的可选头，其结构也不同。标准的可选头长度为24或28字节，通常是28啦。这里就只介绍长度为28的可选头。（因为这种头的长度是自定义的，不同的人定义的结果就不一样。）

这种头的结构如下：

typedef struct {

unsigned short usMagic; // 魔法数字

unsigned short usVersion; // 版本标识

unsigned long ulTextSize; // 正文（text）段大小

unsigned long ulInitDataSZ; // 已初始化数据段大小

unsigned long ulUninitDataSZ; // 未初始化数据段大小

unsigned long ulEntry; //入口点

unsigned long ulTextBase; // 正文段基址

unsigned long ulDataBase; //数据段基址（在PE32中才有）

} OPTHDR;

第一个成员usMagic还是魔法数字，不过这回它的值应该为0x010b或0x0107。当值为0x010b时，说明COFF文件是一个一般的可执行文件；当值为，0x0107时，COFF则为一个ROM镜像文件。

usVersion是COFF文件的版本，ulTextSize是这个可执行COFF的正文段长度，ulInitDataSZ和ulUninitDataSZ分别为已初始化数据段和未初始化数据段的长度。

ulEntry是程序的入口点，也就是COFF载入内存时正文段的位置（EIP寄存器的值），当COFF文件是一个动态库时，入口点也就是动态库的入口函数。

ulTextBase是正文段的基址。

ulDataBase是数据段基址。

其实在这些成员中，只要注意usMagic和ulEntry就可以了。

段落头

段落头紧跟在可选头的后面（如果可选头的长度为0，那么它就是紧跟在文件头后）。它的长度为40 个字节，如下：

typedef struct {

char cName[8]; //8 段名

unsigned long ulVSize; //4 虚拟大小

unsigned long ulVAddr; //4 虚拟地址

unsigned long ulSize; //4 段长度

unsigned long ulSecOffset; //4 段数据偏移

unsigned long ulRelOffset; //4 段重定位表偏移

unsigned long ulLNOffset; //4 行号表偏移

unsigned short usNumRel; //2 重定位表长度

unsigned short usNumLN; //2 行号表长度

unsigned long ulFlags; //4 段标识

} SECHDR;

这个头可是个重要的头头，我们要用到的最终信息就由它来描述。一个COFF文件可以不要其它的节，但文件头和段落头这两节是必不可少的。

cName用来保存段名，常用的段名有.text，.data，.comment，.bss等。.text段是正文段，通常也就是代码段；.data是数据段，在这个数据段中所保存的数据是初始化过的数据；.bss段也可以用来保存数据，不过这里的数据是未初始化的，这个段也是一个空段；.comment段，看名字也知道，它是注释段，用来保存一些编译信息，算是对COFF文件的注释。

ulVSize是段数据载入内存时的大小。只在可执行文件中有效，在目标文件中总为0。如果它的长度大于段的实际长度，则多的部分将用0来填充。

ulVAddr是段数据载入或连接时的虚拟地址。对于可执行文件来说，这个地址是相对于它的地址空间而言。当可执行文件被载入内存时，这个地址就是段中数据的第一个字节的位置。而对于目标文件而言，这只是重定位时，段数据当前位置的一个偏移量。为了计算方便，便定位的计算简化，它通常设为0。

ulSize这才是段中数据的实际长度，也就是段数据的长度，在读取段数据时就由它来确定要读多少字节。

ulSecOffset是段数据在COFF文件中的偏移量。

ulRelOffset是该段的重定位信息的偏移量。它指向了重定位表的一个记录。

ulLNOffset是该段的行号表的偏移量。它指向的是行号表中的一个记录。

usNumRel是重定位信息的记录数。从ulRelOffset指向的记录开始，到第ulNumRel个记录为止，都是该段的重定位信息。

usNumLN和usNumRel相似。不过它是行号信息的记录数。

ulFlags是该段的属性标识。其值如下表：

值 名称 说明

0x0020 STYP\_TEXT 正文段标识，说明该段是代码。

0x0040 STYP\_DATA数据段标识，有些标识的段将用来保存已初始化数据。

0x0080 STYP\_BSS 有这个标识段也是用来保存数据，不过这里的数据是未初始化数据。

注意，在BSS段中，ulVSize、ulVAddr、ulSize、ulSecOffset、ulRelOffset、ulLNOffset、usNumRel、usNumLN的值都为0。（上表只是部分值，其它值在PE格式中介绍，后同）

段数据

是保存各个段的数据的位置。不同类型的段，数据的内容、结构也不尽相同。但在目标文件中，这些数据都是原始数据（Raw Data）。不存在什么特别的格式。

重定位表

这个表所保存的是各个段的重定位信息。这是一张很大的表，因为所有段的重定位信息都在这个表里。各个段落头记录了自己的重定位信息的偏移和数量。要用到重定位信息时就到这个表里来读。当然，你也可以把整个重定位表看成多个重定位表，每个段落都有一个自己的重定位表。这个表只在目标文件中有，可执行文件中是不存在这个表的。

既然有表，那么就会有记录。重定位表中的每一条记录就是一条重定位信息。这个记录的结构很简单，如下：

typedef struct {

unsigned long ulAddr; // 定位偏移 0x05

unsigned long ulSymbol; // 符号

unsigned short usType; // 定位类型

} RELOC;

一共三个成员！ulAddr是要定位的内容在段内偏移。比如：一个正文段，起始位置为0x010，ulAddr的值为0x05，那你的定位信息就要写在0x15处。而且信息的长度要看你的代码的类型，32位的代码要写4个字节，16位的就只要字2个字节。

ulSymbol是符号索引，它指向符号表中的一个记录。注意，这里是索引，不是偏移！它只是符号表中的一个记录的记录号。这个成员指明了重定位信息所对映的符号。

usType是重定位类型的标识。32位代码中，通常只用两种定位方式。一是绝对定位，二是相对定位。其代码如下：

值 名称 说明

6 RELOC\_ADDR32 32位绝对定位。

20 RELOC\_REL32 32位相对定位。

对于不同的处理器，这些值也不尽相同。这里给出的是i386平台上最常用的两个种定位方式的标识。

其定位方式如下：

绝对定位

在绝对定位方式下，你要给出符号的绝对地址（注意，有时候这里可能不是地址，而是值，对于常量来说，你不用给出它的地值，只用给出它的值）。当然，这个地址也不是现成的，你要用符号的相对地址+它所在段的相对地址来得到它的绝对地址。

公式：符号绝对地址=段偏移+符号偏移

这些偏移量你要分别从段落头和符号表中得到。当段落要重定位时，当然还要先重定位段落，才能定位其中的符号。

相对定位

相对定位要复杂一些。它所要的地址信息是相对于当前位置的偏移，这个当前位置就是ulAddr所指向的这个偏移的绝对地址后四个字节（32位代码是四个字节，16位是两个字节）的位置。也就是用定位偏移+当前段偏移+机器字长÷8

公式：当前地址=定位偏移+当前段偏移+机器字长÷8

有了当前地址，相对地址就好计算了。只要用符号的绝对地址减去当前地址就可以了。

公式：相对地址=符号绝对地址-当前地址

计算好了地址，把它写到ulAddr所指向的位置，就一切OK！你已经完成了重定位的工作了。

行号表

行号表在调试时很有用。它把可执行的二进制代码与源代码的行号之间建立了对映关系。这样，当程序执行不正确时（其实正确的也可以J），我们就可以根据当前执行代码的位置得知出错源代码的行号，再加以修改。如果没有它的话，鬼才知道是哪一行出了毛病！

它的格式也很简单。只有两个成员，如下：

typedef struct {

unsigned long ulAddrORSymbol; // 代码地址或符号索引

unsigned short usLineNo; // 行号

} LINENO;

让我们先看第二个成员，usLineNo。这是一个从1开始计数的计数器，它代表源代码的行号。第一个成员ulAddrORSymbol在行号大于0时，代表源代码的地址；而当行号为0时，它就成了行号所对映的符号在符号表中的索引。下面让我们来看看符号表吧！

符号表

符号表是对象文件中用来保存符号信息的一张表，也是COFF文件中最为复杂的一张表。所有段落使用到的符号都在这个表里。它也是由很多条记录组成，每条记录都以如下结构保存：

typedef struct {

union {

char cName[8]; //8 符号名称

struct {

unsigned long ulZero; //字符串表标识

unsigned long ulOffset; // 字符串偏移

} e;

} e;

unsigned long ulValue; //4 符号值

short iSection; //2 符号所在段

unsigned short usType; //2 符号类型

unsigned char usClass; //1 符号存储类型

unsigned char usNumAux; //1 符号附加记录数

} SYMENT;

cName符号名称，和前面所有的名称一样，它也是8个字节，但不同的是它在一个联合体中。和它占相同的存储空间的还有ulZero和ulOffset这两个成员。如果符号的名称只有8个字符，那很好，可以直接放到这个cName中；可是，如果名称的长度大于8个字节，这里就放不下了，只好放到字符串表中。这时候，ulZero的值就会为0，而在ulOffset中会给出我们所用的符号的名称在字符串表中的偏移。

一个符号有了名称不够，它还要有值！ulValue就是这个符号所代表的值。

iSection成员指出了这个符号所在的段落。如果它的值为0，那么这个符号就是一个外部符号，要从其它的COFF文件中解析（连接多个目标文件就是要解析这种符号）。当它的值为-1时，说明这个符号的值是一个常量，不是它在段落中的偏移。而当它的值为-2时，这个符号只是一个调试符号，只有在调试时才会用到它。当它大于0时，才是符号所在段的索引值。

usType是符号的类型标识。它用来说明这个符号的类型，是函数？整型？还是其它什么。这个标识是两个字节。

低字节的低四位是基本标识，它指出了符号的基本类型，如整型，字符，结构，联合等。高四位指出了符号的高级类型，如指针（0001b），函数（0010b），数组（0011b），无类型（0000b）等。编译器，通常不使用基本类型，只使用高级类型。所以，符号的基本类型通常被设为0。

高字节通常未用。

usClass是符号的存储类型标识。它指明了符号的存储方式。

其值与意义见下表：

值 名称 说明

NULL 0 无存储类型。

AUTOMATIC 1 自动类型。通常是在栈中分配的变量。

EXTERNAL 2 外部符号。当为外部符号时，iSection的值应该为0，如果不为0，则ulValue为符号在段中的偏移。

STATIC 3 静态类型。ulValue为符号在段中的偏移。如果偏移为0，那么这个符号代表段名。

REGISTER 4寄存器变量。

MEMBER\_OF\_STRUCT 8 结构成员。ulValue值为该符号在结构中的顺序。

STRUCT\_TAG 10 结构标识符。

MEMBER\_OF\_UNION 11 联合成员。ulValue值为该符号在联合中的顺序。

UNION\_TAG 12 联合标识符。

TYPE\_DEFINITION 13 类型定义。

FUNCTION 101 函数名。

FILE 102 文件名。

最后一个成员usNumAux是附加记录的数量。附加记录是用来描述符号的一些附加信息，为了便于保存，这些附加记录通常选择成为一条符号信息记录的整数倍（多数为1）。所以，如果这个成员的值为1，那么就表示在当前符号信息记录后附加了一条记录，用来保存附加信息。

附加信息的结构是与符号的类型以及存储类型相关的。不同的类型的符号，其附加信息（如果有的话）的结构也不同。如果你不在意这些内容，也可以把它们乎略。

当段的类型为FILE时，附加信息就是一个字符串，它是目标文件对应源文件的名称。其它类型在介绍PE时再进行详细讨论。

字符串表

是用来保存字符串的。它紧接在符号表后。至于为什么要保存字符串，前面已经说过了。这里就不再多说了，只说说字符串的保存格式。

字符串表是所有节中最简单一节。如下图：

0 4

字符串表长度 字符串1\0

.... 字符串n\0

字符串表的前四个字节是字符串表的长度，以字节为单位。其后就是以0结尾的字符串（C风格字符串）。要注意的是，字符串表的长度不仅仅是字符串的长度（这个长度要包括每个字符串后的‘\0’）的总合，它还包括这个长度域的四个字节。符号表中ulOffset成员所指出的偏移就是从字符串表起始处的偏移。比如：指向第一个字符串的符号，ulOffset的值总为4。

下面给出的代码，是从字符串表中读取字符串的典型C代码。

int iStrlen,iCur=4; // iStrLen是字符串表的长度，iCur是当前字符串偏移

char \*str; // 字符串表

read(fn, &iStrlen, 4); // 得到字符串表长度

str = (char \*)malloc(iStrlen); // 为字符串表分配空间

while (iCur<iStrlen ) // 读字符串表，直到全部读入内存

iCur+=read(fn, str+iCur, iStrlen- iCur);

iCur=4; // 把当前字符串偏移指向第一个字符串

while (iCur<iStrlen ) { // 显示每一个字符串

printf("String offset 0x%04X : %s\n", iCur, str + iCur);

iCur+=(strlen(str+iCur)+1); // 计算偏移时不要忘了计算‘\0’字符所占的1个字节！

}

free(str); // 释放字符串表空间

直到这里，整个COFF的结构已经全部介绍完了。标准的COFF文件只有这么多的东西。但MS为了和DOS的可执行文件兼容，以及对可执行文件功能的扩展，在COFF格式中加了很多它自己的内容。

格式文件

Microchip 的COFF 规范基于Understanding and Using COFF（Gintaras R. Gircys &copy;1988, O’Reilly and Associates, Inc）中描述的UNIX&reg; System V COFF 格式。但Microchip COFF 格式和UNIX SystemV COFF 格式有不同之处。详细信息可查阅《MPLAB C18用户指南》。

COFF Spec

This document should be considered to be the ultimate reference to the COFF format. That doesn't mean it's complete, but since this format isn't really documented elsewhere, this is as good as it gets.

All programs reading COFF files should include <coff.h>

Note: Unless otherwise specified, all numeric fields are stored in host native order, which is LSB-first (little endian), and all file offsets are relative to the beginning of the COFF object (i.e. the file header is always at offset zero, even when the object is inside a library).

Structure Located? Purpose

File Header Beginning of file Overview of the file; controls layout of other sections

Optional Header Follows file header For executables, used to store the initial %eip

Section Header Follow optional header; count determined by file header Maintain location and size information about code and data sections

Section Data Stored in section header Contains code and data for the program

Relocation Directives Stored in section header Contain fixup information needed when relocating a section

Line Numbers Stored in section header Hold address of each line number in code/data sections

Symbol Table Stored in file header Contains one entry for each symbol this file defines or references

String Table Follows symbol table Stores symbol names; first four bytes are total length

COFF: File Header

typedef struct {

unsigned short f\_magic; /\* magic number \*/

unsigned short f\_nscns; /\* number of sections \*/

unsigned long f\_timdat; /\* time & date stamp \*/

unsigned long f\_symptr; /\* file pointer to symtab \*/

unsigned long f\_nsyms; /\* number of symtab entries \*/

unsigned short f\_opthdr; /\* sizeof(optional hdr) \*/

unsigned short f\_flags; /\* flags \*/

} FILHDR;

This structure always exists at the beginning of the COFF object. When reading this header, you should read FILHSZ bytes, and not rely on sizeof(FILHDR) to give the correct size.

f\_magic - magic number

This is a constant value for all COFF files, and is used to detect the fact that the file is COFF. The value of this field must be I386MAGIC (0x14c) and is stored in little-endian format, so the first two bytes of any COFF file are 0x4c and 0x01.

f\_nscns - number of sections

The number of sections (and thus section headers) contained within this file.

f\_timdat - file time & date stamp

The time that this coff file was created. The value has the same meaning as the time\_t type.

f\_symptr - symbol table pointer

Contains the file offset of the symbol table.

f\_nsyms - number of symbols in the symbol table

The number of symbols in the symbol table.

f\_opthdr - optional header size

The number of extra bytes that follow the file header, before the section headers begin. Often used to store the optional a.out header. Regardless of what optional header you expect, you should read (or skip) exactly the number of bytes given in this field before reading the section headers.

f\_flags - flag bits

These flags provide additional information about the state of this coff object. The flags are as follows:

Bit

Symbol

Meaning

0x0001 F\_RELFLG If set, there is no relocation information in this file. This is usually clear for objects and set for executables.

0x0002 F\_EXEC If set, all unresolved symbols have been resolved and the file may be considered executable.

0x0004 F\_LNNO If set, all line number information has been removed from the file (or was never added in the first place).

0x0008 F\_LSYMS If set, all the local symbols have been removed from the file (or were never added in the first place).

0x0100 F\_AR32WR Indicates that the file is 32-bit little endian

COFF: Optional Header

The optional header immediately follows the file header in the COFF file. The size of this header is stored in the f\_opthdr field of the file header. You must read that many bytes from the file regardless of how big you expect the optional header to be.

Two optional headers are defined for objects:

Struct

Size

Purpose

AOUTHDR 28 Added to executables to provide the entry point of the program

GNU\_AOUT 32 Unknown

typedef struct {

unsigned short magic; /\* type of file \*/

unsigned short vstamp; /\* version stamp \*/

unsigned long tsize; /\* text size in bytes, padded to FW bdry\*/

unsigned long dsize; /\* initialized data " " \*/

unsigned long bsize; /\* uninitialized data " " \*/

unsigned long entry; /\* entry pt. \*/

unsigned long text\_start; /\* base of text used for this file \*/

unsigned long data\_start; /\* base of data used for this file \*/

} AOUTHDR;

The only two fields you should rely on are described below.

magic - magic number

Always the value ZMAGIC (0x010b).

entry - entry point

This should be used to provide the initial value of %eip when the program is initialized.

COFF: Section Header

typedef struct {

char s\_name[8]; /\* section name \*/

unsigned long s\_paddr; /\* physical address, aliased s\_nlib \*/

unsigned long s\_vaddr; /\* virtual address \*/

unsigned long s\_size; /\* section size \*/

unsigned long s\_scnptr; /\* file ptr to raw data for section \*/

unsigned long s\_relptr; /\* file ptr to relocation \*/

unsigned long s\_lnnoptr; /\* file ptr to line numbers \*/

unsigned short s\_nreloc; /\* number of relocation entries \*/

unsigned short s\_nlnno; /\* number of line number entries \*/

unsigned long s\_flags; /\* flags \*/

} SCNHDR;

This structure always exists immediately following any optional header in the COFF file (or following the file header, if f\_opthdr is zero). When reading this header, you should read SCNHSZ bytes, and not rely on sizeof(SCNHDR) to give the correct size. The number of section headers present is given in the f\_nscns field of the file header.

s\_name - section name

The name of the section. The section name will never be more than eight characters, but be careful to handle the case where it's exactly eight characters - there will be no trailing null in the file! For shorter names, there field is padded with null bytes.

s\_paddr - physical address of section data

This is the address at which the section data should be loaded into memory. For linked executables, this is the absolute address within the program space. For unlinked objects, this address is relative to the object's address space (i.e. the first section is always at offset zero).

s\_vaddr - virtual address of section data

Always the same value as s\_paddr.

s\_size - section data size

The number of bytes of data stored in the file for this section. You should always read this many bytes from the file, beginning s\_scnptr bytes from the beginning of the object.

s\_scnptr - section data pointer

This contains the file offset of the section data.

s\_relptr - relocation data pointer

The file offset of the relocation entries for this section.

s\_lnnoptr - line number table pointer

The file offset of the line number entries for this section.

s\_nreloc - number of relocation entries

The number of relocation entries for this section. Beware files with more than 65535 entries; this field truncates the value with no other way to get the "real" value.

s\_nlnno - number of line number entries

The number of line number entries for this section. Beware files with more than 65535 entries; this field truncates the value with no other way to get the "real" value.

s\_flags - flag bits

These flags provide additional information for each section. Flags other than those set below may be set, but are of no use aside from what these three provide.

Bit

Symbol

Meaning

0x0020 STYP\_TEXT If set, indicates that this section contains only executable code.

0x0040 STYP\_DATA If set, indicates that this section contains only initialized data.

0x0080 STYP\_BSS If set, indicates that this section defines uninitialized data, and has no data stored in the coff file for it.

COFF: Relocation Directives

typedef struct {

unsigned long r\_vaddr; /\* address of relocation \*/

unsigned long r\_symndx; /\* symbol we're adjusting for \*/

unsigned short r\_type; /\* type of relocation \*/

} RELOC;

Warning: This structure's size is not a multiple of four. When reading from file, it is strongly recommended that either (1) you read each entry in a loop, reading RELSZ bytes each time, or allocate a block of memory and calculate a pointer to each entry you need by multiplying by RELSZ. In no case should you assume that array addressing or sizeof(RELOC) will be useful.

There are only two types of relocations that you will encounter in a normal COFF object.

Type

Value

Purpose

RELOC\_ADDR32 6 Relocate a 32-bit absolute reference

RELOC\_REL32 20 Relocate a 32-bit relative reference

For any relocation, you must determine the new address of the relocated symbol that we are adjusting for. If the symbol is in another object (external), the symbol table will contain a reference to that external symbol, and the relocation will refer to that symbol table entry. If the symbol is in the same object, the symbol table will have entries that refer to the sections themselves (always there and always private) that will be referred to. When you relocate the section itself, these symbols will reflect its new location.

RELOC\_ADDR32

To do this relocation, you must perform the following steps:

Get the address of the symbol referred to.

Add the value currently stored in the location being adjusted.

Store the value back into the location being adjusted.

RELOC\_REL32

This relocation happens normally only in executable sections, and refers only to external symbols. To do this relocation, you must perform the following steps:

Get the address of the symbol referred to.

Add the value currently stored in the location being adjusted.

Subtract the address of the beginning of the section.

Add the original (unrelocated) address of the beginning of this section. Normally this is zero for as only the \_text section, which is first (and thus at unrelocated address zero), has relative relocs.

Note: The preceeding two steps can be replaced with the single step of "subtract the amount you moved this section".

Store the value back into the location being adjusted.

COFF: Line Numbers

typedef struct {

union {

unsigned long l\_symndx; /\* function name symbol index \*/

unsigned long l\_paddr; /\* address of line number \*/

} l\_addr;

unsigned short l\_lnno; /\* line number \*/

} LINENO;

Warning: This structure's size is not a multiple of four. When reading from file, it is strongly recommended that either (1) you read each entry in a loop, reading LINESZ bytes each time, or allocate a block of memory and calculate a pointer to each entry you need by multiplying by LINESZ. In no case should you assume that array addressing or sizeof(LINENO) will be useful.

Each executable section has its own line number table. Each function in that section is numbered independently, with the start of the function (the line with the opening brace) numbered as line one for that function. Each function in the line number table will have one entry where l\_lnno is zero and the symbol table entry for the function is in l\_symndx. This entry is followed by entries for each line of the function, with l\_lnno set to the function-relative line number (1..N) and l\_paddr set to the address of the first assembler codes for that line.

To figure out absolute line numbers, you must look in the symbol table for the function, find the "beginning of function" symbol (type C\_FCN, usually right after the function's C\_EXT or C\_STAT symbol) where the absolute line number for the function (equivalent to line one in the line number table's scheme) is stored (in AUXENT.x\_sym.x\_misc.x\_lnsz.x\_lnno), and add that to the relative line numbers in the table.

The trick to getting line numbers right is to remember that the lines of the source file start at one (the first line in the file is line one) and functions are numbered starting at one also. When you add them up, you get one too many ones, so you must then subtract one to get the right line number.

COFF: Symbol Table

typedef struct {

union {

char e\_name[E\_SYMNMLEN];

struct {

unsigned long e\_zeroes;

unsigned long e\_offset;

} e;

} e;

unsigned long e\_value;

short e\_scnum;

unsigned short e\_type;

unsigned char e\_sclass;

unsigned char e\_numaux;

} SYMENT;

The symbol table is probably one of the most complex parts of the COFF object, mostly because there are so many symbol types. The symbol table has entries for all symbols and meta-symbols, including public, static, external, section, and debugging symbols.

e.e\_name - inlined symbol name

If the symbol's name is eight characters or less, it is stored in this field. Note that the first character overlaps the e\_zeroes field - by doing so, the e\_zeroes field can be used to determine if the symbol name has been inlined. Beware that the name is null terminated only if it is less than eight characters long, else it is not null terminated.

e.e.e\_zeroes - flag to tell if name is inlined

If this field is zero, then the symbol name is found by using e\_offset as an offset into the string table. If it is nonzero, then the name is in the e\_name field.

e.e.e\_offset - offset of name in string table

If e\_zeroes is zero, this field contains the offset of the symbol name in the string table.

e\_value - the value of the symbol

The value of the symbol. For example, if the symbol represents a function, this contains the address of the function. The meaning of the value depends on the type of symbol (below).

e\_scnum - section number

The number of the section that this symbol belongs to. The first section in the section table is section one. In addition, e\_scnum may be one of the following values:

Symbol

Value

Meaning

N\_UNDEF 0 An undefined (extern) symbol

N\_ABS -1 An absolute symbol (e\_value is a constant, not an address)

N\_DEBUG -2 A debugging symbol

e\_type - symbol type

The type of the symbol. This is made up of a base type and a derived type. For example, "pointer to int" is "pointer to T" and "int".

Type

Bits

Meaning

T\_NULL ---- 0000 No symbol

T\_VOID ---- 0001 void function argument (not used)

T\_CHAR ---- 0010 character

T\_SHORT ---- 0011 short integer

T\_INT ---- 0100 integer

T\_LONG ---- 0101 long integer

T\_FLOAT ---- 0110 floating point

T\_DOUBLE ---- 0111 double precision float

T\_STRUCT ---- 1000 structure

T\_UNION ---- 1001 union

T\_ENUM ---- 1010 enumeration

T\_MOE ---- 1011 member of enumeration

T\_UCHAR ---- 1100 unsigned character

T\_USHORT ---- 1101 unsigned short

T\_UINT ---- 1110 unsigned integer

T\_ULONG ---- 1111 unsigned long

T\_LNGDBL ---1 0000 long double (special case bit pattern)

DT\_NON --00 ---- No derived type

DT\_PTR --01 ---- pointer to T

DT\_FCN --10 ---- function returning T

DT\_ARY --11 ---- array of T

The BTYPE(x) macro extracts the base type from e\_type. Note that all DT\_\* must be shifted by N\_BTSHIFT to get actual values, as in:

e\_type = base + derived << N\_BTSHIFT;

There are also macros ISPTR, ISFCN, and ISARY that test the upper bits for the derived type.

e\_sclass - storage class

This tells where and what the symbol represents.

Class

Value

Meaning

C\_NULL 0 No entry

C\_AUTO 1 Automatic variable

C\_EXT 2 External (public) symbol - this covers globals and externs

C\_STAT 3 static (private) symbol

C\_REG 4 register variable

C\_EXTDEF 5 External definition

C\_LABEL 6 label

C\_ULABEL 7 undefined label

C\_MOS 8 member of structure

C\_ARG 9 function argument

C\_STRTAG 10 structure tag

C\_MOU 11 member of union

C\_UNTAG 12 union tag

C\_TPDEF 13 type definition

C\_USTATIC 14 undefined static

C\_ENTAG 15 enumaration tag

C\_MOE 16 member of enumeration

C\_REGPARM 17 register parameter

C\_FIELD 18 bit field

C\_AUTOARG 19 auto argument

C\_LASTENT 20 dummy entry (end of block)

C\_BLOCK 100 ".bb" or ".eb" - beginning or end of block

C\_FCN 101 ".bf" or ".ef" - beginning or end of function

C\_EOS 102 end of structure

C\_FILE 103 file name

C\_LINE 104 line number, reformatted as symbol

C\_ALIAS 105 duplicate tag

C\_HIDDEN 106 ext symbol in dmert public lib

C\_EFCN 255 physical end of function

e\_numaux - number of auxiliary entries

Each symbol is allowed to have additional data that follows it in the symbol table. This field tells how many equivalent SYMENTs are used for aux entries. For most symbols, this is zero. A value of one allows up to SYMESZ bytes of auxiliary information for that symbol. A non-exhaustive list of auxiliary entries follows, based on the storage class (e\_sclass) or type (e\_type) of the symbol.

Auxiliary Entries

DT\_ARY

.x\_sym.x\_misc.x\_lnsz.x\_size

size in bytes (size\*count)

T\_STRUCT

T\_UNION

T\_ENUM

.x\_sym.x\_tagndx

syment index for list of tags (will point to C\_STRTAG, C\_UNTAG, or C\_ENTAG)

.x\_sym.x\_misc.x\_lnsz.x\_size

size in bytes (size\*count)

T\_NULL | C\_STAT - section symbols (like .text)

.x\_scn.x\_scnlen

section length (bytes)

.x\_scn.x\_nreloc

number of relocation entries (ushort)

.x\_scn.x\_nlinno

number of line numbers (ushort)

C\_STRTAG - will be followed by C\_MOS's and C\_EOS

C\_UNTAG - will be followed by C\_MOU's and C\_EOS

C\_ENTAG - will be followed by C\_MOE's and C\_EOS

.x\_sym.x\_x\_misc.x\_lnsz.x\_size

The size of the struct/union/enum

.x\_sym.x\_fcnary.x\_fcn.x\_endndx

The symbol index after our list.

C\_EOS

.x\_sym.x\_misc.x\_lnsz.x\_size

the size of the struct/union/enum

.x\_sym.x\_tagndx

The symbol index of the start of our list.

C\_FIELD

.x\_sym.x\_x\_misc.x\_lnsz.x\_size

the number of bits

C\_BLOCK

.x\_sym.x\_misc.x\_lnsz.x\_lnno

starting line number

.x\_sym.x\_fcnary.x\_fcn.x\_endndx

The symbol index after our block (if .bb)

C\_FCN

.x\_sym.x\_misc.x\_lnsz.x\_lnno

starting line number

.x\_sym.x\_misc.x\_lnsz.x\_size

size in bytes

C\_FILE

.x\_file.x\_fname

.x\_file.x\_n.x\_zeroes

.x\_file.x\_n.x\_offset

These three specify the file name, just like the three fields used to specify the symbol name.

Meanings of the Values

SClass

Meaning of the Value

C\_AUTO

C\_ARG Address of the variable, relative to %ebp

C\_EXT

C\_STAT

others The address of the symbol

C\_REG The register number assigned to this variable

C\_MOS Offset of the member from the beginning of the structure

C\_MOE The value of this enum member

C\_FIELD The mask for this field

C\_EOS size of struct/union/enum

COFF: String Table

The string table contains the names of symbols that are too long to inline in the symbol table. To read the string table, position the file pointer just after the symbol table (usually, you read the strings right after you read the symbols anyway), and read four bytes as one 32-bit little endian integer. Allocate this much memory. Set the first four bytes of the memory to zero, and read the remainder of the string table (length-4) into the remainder of the memory (ptr+4). A code sample would look like this:

int i;

char \*s;

read(fd, &i, 4);

s = (char \*)malloc(i);

memset(s, 0, 4);

read(fd, s+4, i-4);

All references to strings in this table are offsets from the beginning of this memory block. Note that offsets of zero are legal and will result in a zero-length string because of those four zeros you put at the beginning (where the length used to be).