XXXX notes XXXX add stuff about object format design, card decks where the order didn't matter much, the ways that object files limited language semantics, e.g., limited symbol lengths. say something about IBM object decks

Object Files

THIS CHAPTER WILL BE REVISED AND CORRECTED. SOME OF THE MATERIAL ON WINDOWS OBJECT FILES IS JUST WRONG.

Compilers and assemblers create object files containing the generated binary code and data for a source file. Linkers combine multiple object files into one, loaders take object files and load them into memory. In this chapter we delve into the details of object file formats and contents.

What goes into an object file?

An object file contains five kinds of information.

- *Header information:* overall information about the file, such as the size of the code, name of the source file it was translated from, and creation date.
- *Object code:* Binary instructions and data generated by a compiler or assembler.
- *Relocation:* A list of the places in the object code that have to be fixed up when the linker changes the addresses of the object code.
- *Symbols:* Global symbols defined in this module, symbols to be imported from other modules or defined by the linker.
- *Debugging information:* Other information about the object code not needed for linking but of use to a debugger. This includes source file and line number information, local symbols, descriptions of data structures used by the object code such as C structure definitions. (Some object files contain even more than this, but these are plenty to keep us occupied in this chapter.)

Not all object formats contain all of these kinds of information, and it's possible to have quite useful formats with little or no information beyond the object code. We'll tour a series of popular formats, starting with the simplest, and working up to the most complicated.

The null object format: MS-DOS .COM files

It's quite possible to have a usable object file with no information in it whatsoever other than the runnable binary code. The MS-DOS .COM format is the best-known example. A .COM file literally consists of nothing other than binary code. When the operating system runs a .COM file, it merely loads the contents of the file into a chunk of free memory starting at offset 0x100, (0-FF are the, PSP, Program Segment Prefix with command line arguments and other parameters), sets the x86 segment registers all to point to the PSP, the stack pointer to the end of the segment, since the stack grows downward, and jumps to the beginning of the loaded program.

The segmented architecture of the x86 makes this work. Since all x86 program addresses are interpreted relative to the base of the current segment, and the operating system sets all of the segment registers to point to the place where the file is loaded, the program is always loaded at segment-relative location 0x100. Hence, for a program that fits in a single segment, no fixups are needed.

For programs that don't fit in a single segment, the fixups are the programmer's problem, and there are indeed programs that start out by fetching one of their segment registers, and adding its contents to stored segment values elsewhere in the program. Of course, this is exactly the sort of tedium that linkers and loaders are intended to automate, and MS-DOS does that with .EXE files, described later in this chapter.

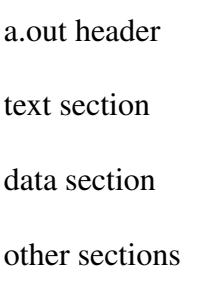
Code sections: Unix a.out files

Computers with hardware memory relocation (nearly all of them, these days) usually create a new process with a clean address space for each newly run program, in which case programs can be linked to start at a fixed address and require no relocation at load time. The Unix a.out object format hanldes this situation.

In the simplest case, an a.out file consists of a small header followed by the executable code (called the text section for historical reasons) and the initial values for static data, Figure 1. The PDP-11 had only 16 bit addressing, which limited programs to a total of 64K. This limit quickly be-

came too small, so later models in the PDP-11 line provided separate address spaces for code (I for Instruction space) and data (D space), so a single program could contain both 64K of code and 64K of data. To support this feature, the compilers, assembler, and linker were modified to create two-section object files, with the code in the first section and the data in the second section, and the program loader loaded the first section into a process' I space and the second into the D space.

Figure 1: Simplifed a.out



Separate I and D space had another performance advantage: since a program couldn't change its own I space, multiple copies of a single program could share a single copy of a program's code, while keeping separate copies of the program's data. On a time-shared system like Unix, multiple copies of the shell (the command interpreter) and network daemons are common, and shared program code saves considerable real memory.

The only currently common computer that still uses separate addressing for code and data is the 286 (or 386 in 16 bit protected mode), but even on more modern machines with large address spaces, shared read-only code offers performance gains, so all modern loaders support it. This means that linker formats must at the least mark read-only versus read-write sections. In practice, most linker formats have many sections, such as read-only data, symbols and relocation for subsequent linking, debugging symbols, and shared library information. (Unix convention confusingly calls the file sections segments, so we use that term in discussions of Unix file

formats.)

a.out headers

The header varies somewhat from one version of Unix to another, but the version in BSD Unix, Figure 2 is typical. (In the examples in this chapter, int values are 32 bits, and short are 16 bits.)

Figure 2: a.out header

```
int       a_magic;  // magic number
int       a_text;             // text segment size
int       a_data;             // initialized data size
int       a_bss;              // uninitialized data size
int       a_syms;             // symbol table size
int       a_entry;  // entry point
int       a_trsize; // text relocation size
int       a_drsize; // data relocation size
```

The magic number indicates what kind of executable file this is. (Historically, the magic number on the original PDP-11 was octal 407, which was a branch instruction that would jump over the next seven words of the header to the beginning of the text segment. That permitted a primitive form of position independent code. A bootstrap loader could load the entire executable including the file header to be loaded by into memory, usually at location zero, and then jump to the beginning of the loaded file to start the program. Only a few standalone programs ever used this ability, but the 407 magic number is still with us 25 years later.) Different magic numbers tell the operating system program loader to load the file in to memory differently; we discuss these variations below. The text and data segment sizes and are the sizes in bytes of the read-only code and read-write data that follow the header. Since Unix automatically initializes newly allocated memory to zero, any data with an initial contents of zero

or whose contents don't matter need not be present in the a.out file. The uninitialized size says how much uninitialized data (really zero-initialized) data logically follows the data in the a.out file.

The field gives the starting address of the program, while and say how much symbol table and relocation information follow the data segment in the file. Programs that have been linked and are ready to run need no symbol nor relocation info, so these fields are zero in runnable files.

Interactions with virtual memory

The process involved when the operating system loads and starts a simple two-segment file is straightforward, Figure 3:

Figure 3: Loading an a.out into a process

picture of file and segments with arrows pointing out data flows

---

- Read the a.out header to< get the segment sizes.
- Check to see if there's already a sharable code segment for this file. If so, map that segment into the process' address space. If not, create one, map it into the address space, and read the text segment from the file into the new memory segment.
- Create a private data segment large enough for the combined data and BSS, map it into the process, and read the data segment from the file into the data segment. Zero out the BSS segment.
- Create and map in a stack segment (usually separate from the data segment, since the data heap and stack grow separately)
- Set registers appropriately and jump to the starting address.

This scheme (known as NMAGIC, where the N means new, as of about 1975) works quite well, and PDP-11 and early VAX Unix systems used it

for years, but when Unix systems gained virtual memory, several improvements to this simple scheme sped up program loading and saved considerable real memory.

On a paging system, the simple scheme above allocates fresh virtual memory for each text segment and data segment. Since the a.out file is already stored on the disk, the object file itself can be mapped into the process' address space. This saves disk space, since new disk space for virtual memory need only be allocated for pages that the program changes, and can speed program startup, since the virtual memory system need only load in from disk the pages that the program's actually using, not the whole file.

A few changes to the a.out format make this possible, Figure 4. (These create what's known as ZMAGIC format.) These changes align the segments in the object file on page boundaries. On systems with 4K pages, the a.out header is expanded to 4K, and the text segment's size is rounded up to the next 4K boundary. There's no need to round up the size of the data segment, since the BSS segment logically follows the data segment, and is zeroed by the program loader anyway.

Figure 4: Mapping an a.out into a process

Picture of file and segments, with page frames mapping into segments

---

ZMAGIC files reduce unneeded paging, but at the cost of wasting a lot of disk space. The a.out header is only 32 bytes long, yet an entire 4K of disk space is allocated. The gap between the text and data also wastes 2K, half a 4K page, on average. Both of these are fixed in the compact pagable format (known as QMAGIC.)

Compact pagable files consider the a.out header to be part of the text segment, since there's no particular reason that the code in the text segment has to start at location zero. Indeed, program zero is a particularly bad place to load a program since uninitialized pointer variables often contain

zero. (An entire generation of UNIX programs contained severe undiagnosed pointer bugs because in the VAX versions of Berkeley Unix location zero was valid and always contained zero.) The code actually starts immediately after the header, and the whole page is mapped into the second page of the process, leaving the first page unmapped so that pointer references to location zero will fail, Figure 5. This has the harmless side-effect of mapping the header into the process as well.

Figure 5: Mapping a compact a.out into a process

Picture of file and segments, with page frames mapping into segments

The text and data segments in a QMAGIC executable are each rounded up to a full page, so the system can easily map file pages to address space pages. The last page of the data segment is padded out with zeros for BSS data; if there is more BSS data than fits in the padding area, the a.out header contains the size of the remaining BSS area to allocate.

Although BSD Unix loads programs at location zero (or 0x1000 for QMAGIC), other versions of Unix load programs at other addresses. For example, System V for the Motorola 68K series loads at 0x80000000, and for the 386 loads at 0x8048000. It doesn't matter where the load address is so long as it's page aligned, and the linker and operating system can permanently agree what it is.

Relocation: MS-DOS EXE files

The a.out format is quite adequate for systems that assign a fresh address space to each process so that every program can be loaded at the same logical address. Many systems are not so fortunate. Some load all the programs into the same address space. Others give each program its own address space, but don't always load the program at the same address. (32 bit versions of Windows fall into this last category.)

In these cases, executable files contain *relocation entries* often called *fixups* that identify the places in the program where addresses need to be modified when the program is loaded. One of the simplest formats with fixups is the MS-DOS EXE format.

As we saw with the .COM format above, DOS loads a program into a contiguous chunk of available real-mode memory. If the program doesn't fit in one segment, the segment numbers in the file have to be fixed up to match the address where the program is actually loaded. The segment numbers are stored as though the program will be loaded at location zero, so the fixup action is to add to every stored segment number the base paragraph number at which the program is actually loaded. The offsets within the segments don't change, because the program is loaded as a block with the relative positions of addresses within the program unchanged.

Each .EXE File starts with a header shown in Figure 6. Following the header is some extra information of variable length (used for overlay loaders, self-extracting archives, and other application-specific hackery) and a list of the fixup addresses in 32 bit segment:offset format. The fixup addresses are relative to the base of the program, so the fixups themselves have to be relocated to find the addresses in the program to change. After the fixups comes the program code. There may be more information, ignored by the program loader, after the code. (In the example below, far pointers are 32 bits with a 16 bit segment number and 16 bit offset.)

Figure 6: Format of .EXE file header

```
char signature[2] = "MZ";     // magic number
short lastsize;     // # bytes used in last block
short nblocks;      // number of 512 byte blocks
short nreloc;       // number of relocation entries
short hdrsize;      // size of file header in 16 byte paragr
short minalloc;     // minimum extra memory to allocate
short maxalloc;     // maximum extra memory to allocate
void far *sp;       // initial stack pointer
short checksum;     // ones complement of file sum
```

```
void far *ip;        // initial instruction pointer
short relocpos;      // location of relocation fixup table
short noverlay;      // Overlay number, 0 for program
char extra[];        // extra material for overlays, etc.
void far *relocs[]; // relocation entries, starts at relocpo
```

Loading an .EXE file is only slightly more complicated than loading a .COM file.

- Read in the header, check the magic number for validity
- Find a suitable area of memory. The and fields say the minimum and maximum number of extra paragraphs of memory to allocate beyond the end of the loaded program. (Linkers almost invariably default the minimum to zero and the maximum to 0xFFFF.)
- Create a PSP, the control area at the head of the program.
- Read in the program code immediately after the PSP. The and fields define the length of the code.
- Start reading fixups at For each fixup, add the base address of the program code to the segment number in the fixup, then use the relocated fixup as a pointer to a program address to which to add the base address of the program code.
- Set the stack pointer to relocated, and jump to relocated, to start the program.

Other than the peculiarities associated with segmented addressing, this is a pretty typical setup for program loading. In a few cases, different pieces of the program are relocated differently. In 286 protected mode, each segment of code or data in the executable file is loaded into a separate segment in the system, but the segment numbers cannot for architectural reasons be consecutive. In that case, the executable has a table near the beginning listing all of the segments that the program will require, the system makes a table of actual segment numbers corresponding to each seg-

ment in the executable. When processing fixups, the system looks up the logical segment number in that table and replaces it with the actual segment number, a process more akin to symbol binding than to relocation.

Some systems permit symbol resolution at load time as well, but we save that topic for Chapter 10.

Symbols and relocation

The object formats we've considered so far are all loadable, that is, they can be loaded into memory and run directly. Most object files aren't loadable, but rather are intermediate files passed from a compiler or assembler to a linker or library manager. These linkable files can be considerably more complex than runnable ones. Runnable files have to be simple enough to run on the ''bare metal'' of the computer, while linkable files are processed by a layer of software which can do very sophisticated processing.

We look at four formats of increasing complexity: relocatable a.out used on BSD UNIX systems, ELF used on System V, ECOFF or PE used on 32 bit Windows, and the OMF linkable format used on pre-ECOFF Windows systems..

Relocatable a.out

Unix systems have always used a single object format for both runnable and linkable files, with the runnable files leaving out the sections of use only to the linker. The a.out format back in Figure 2 includes several fields used by the linker. The sizes of the relocation tables for the text and data segments are in and and the size of the symbol table is in The three sections follow the text and data, Figure 7

Figure 7: Simplifed a.out

a.out header

text section

data section

text relocation

data relocation

symbol table

string table

Relocation entries

Relocation entries serve two functions. When a section of code is relocated to a different base address, relocation entries mark the places in the code that have to be modified. In a linkable file, there will also be relocation entries marking references to undefined symbols, so the linker knows where to patch in the symbol's value when the symbol is finally defined.

Figure 8 shows the format of a relocation entry. Each entry contains the address within the text or data section to be relocated, along with information that defines what to do. The address is the offset from the beginning of the text or data segment of a relocatable item. The length field says how long the item is, values 0 through three mean 1, 2, 4, or (on some architectures) 8 bytes. The pcrel flag means that this is a ‘‘PC relative’’ item, that is, it's used in an instruction as a relative address.

Figure 8: Relocation entry format

Draw this with boxes

-- four byte address

-- three byte index, one bit pcrel flag, 2 bit length field, one bit extern flag, four spare bits

The extern flag controls the interpretation of the index field. If the extern flag is off, this is a plain relocation item, and the index tells which segment (text, data, or BSS) the item is addressing. If the extern flag is on, this is a reference to an external symbol, and the index is the symbol number in the file's symbol table.

This relocation format is adequate for most machine architectures, but some of the more complex ones need extra flag bits to indicate, e.g., three-byte 370 address constants or high and low half constants on the MIPS.

Symbols and strings

The final section of an a.out file is the symbol table. Each entry is 12 bytes and describes a single symbol, Figure 9.

Figure 9: Symbol format

Draw this with boxes, too:

- four byte name offset

- one byte type

- one spare byte

- two byte debugger info

- four byte value

Unix compilers permit arbitrarily long identifiers, so the name strings are all in a string table that follows the symbol table. The first item in a symbol table entry is the offset in the string table of the null-terminated name of the symbol. In the type byte, if the low bit is set the symbol is external (visible to other modules). Non-external symbols are not needed for linking but can be used by debuggers. The rest of the bits are the symbol type.

The most important types include:

- *text*, *data*, or *bss*: A symbol defined in this module. External bit may or may not be on. Value is the relocatable address in the module corresponding to the symbol.
- *abs*: An absolute non-relocatable symbol. (Rare outside of debugger info.) External bit may or may not be on. Value is the absolute value of the symbol.
- *undefined*: A symbol not defined in this module. External bit must be on. Value is usually zero, but see the ‘‘common block hack’’ below. These symbol types are adequate for older languages such as C and Fortran and, just barely, for C++.

As a special case, if an undefined external symbol has a non-zero value, that value is a hint to the linker how large a block of storage the program expects the symbol to address. At link time, if there is no definition of the symbol, the linker creates a block of storage by that name in the BSS segment with the size being the largest hint value found in any of the linked modules. This ‘‘common block hack’’supports typical (albeit non standard conformant) usage of Fortran common blocks and uninitialized C external data.

a.out summary

The a.out format is a simple and effective one for relatively simple systems with paging. It has fallen out of favor because it doesn't easily support for dynamic linking. Also, a.out doesn't support C++, which requires special treatment of initializer and finalizer code, very well.

Unix ELF

The traditional a.out format served the Unix community for over a decade, but with the advent of Unix System V, AT&T decided that it needed something better to support cross-compilation, dynamic linking and other modern system features. Early versions of System V used COFF, Common Object File Format, which was originally intended for cross-compiled embedded systems and didn't work all that well for a time-sharing system. In later versions of System V, COFF was superseded by ELF, Executable and

Linking Format. ELF has been adopted by the popular freeware Linux variant of Unix as well. ELF has an associated debugging format called DWARF which we visit in Chapter 5.

ELF files come in three slightly different flavors: relocatable, executable, and shared object. Relocatable files are created by compilers and assemblers but need to be processed by the linker before running. Executable files have all relocation done and all symbols resolved except perhaps shared library symbols to be resolved at runtime. Shared objects are shared libraries, containing both symbol information for the linker and directly runnable code for runtime.

ELF files have an unusual dual nature, Figure 10. Compilers, assemblers, and linkers treat the file as a set of logical sections described by a section header table, while the system loader treats the file as a set of segments described by a program header table. A single segment will usually consist of several sections. For example, a ‘‘loadable read-only’’ segment could contain sections for executable code, read-only data, and symbols for the dynamic linker. Relocatable files have section tables, executable files have program header tables, and shared objects have both. The sections are intended for further processing by a linker, while the segments are intended to be mapped into memory.

Figure 10: Two views of an ELF file

linking view and execution view, adapted from fig 1-1 in Intel TIS document

---

ELF files all start with the ELF header, Figure 11. The header is designed to be decodable even on machines with a different byte order from the file's target architecture. The first four bytes are the magic number identifying an ELF file, followed by three bytes describing the format of the rest of the header. Once a program has read the and flags, it knows the byte order and word size of the file and can do the necessary byte swapping and size conversions. Other fields provide the size and location of the section

header and program header, if present,

Figure 11: ELF header

```
char magic[4] = "\177ELF";    // magic number
char class;          // address size, 1 = 32 bit, 2 = 64 bit
char byteorder;      // 1 = little-endian, 2 = big-endian
char version;        // header version, always 1
char pad[9];

short filetype;      // file type: 1 = relocatable, 2 = execu
                     // 3 = shared object, 4 = core image
short archtype;      // 2 = SPARC, 3 = x86, 4 = 68K, etc.
int version;         // file version, always 1
int entry;           // entry point if executable
int phdrpos;         // file position of program header or 0
int shdrpos;         // file position of section header or 0
int flags;           // architecture specific flags, usually
short hdrsize;       // size of this ELF header
short phdrent;       // size of an entry in program header
short phdrcnt;       // number of entries in program header o
short shdrent;       // size of an entry in section header
short phdrcnt;       // number of entries in section header o
short strsec;        // section number that contains section
```

Relocatable files

A relocatable or shared object file is considered to be a collection of sections. Each section contains a single type of information, such as program code, read-only or read-write data, relocation entries, or symbols. Every symbol defined in the module is defined relative to a section, so a procedure's entry point would be relative to the program code section that contains that procedure's code. There are also two pseudo-sections (number

0xfff1) which logically contains absolute non-relocatable symbols, and (number 0xfff2) that contains uninitialized data blocks, the descendant of the a.out common block hack, and section zero is always a null section, with an all-zero section table entry.

Figure 11: Section header

```
int sh_name;        // name, index into the string table
int sh_type;        // section type
int sh_flags;       // flag bits, below
int sh_addr;        // base memory address, if loadable, or
int sh_offset;      // file position of beginning of section
int sh_size;        // size in bytes
int sh_link;        // section number with related info or z
int sh_info;        // more section-specific info
int sh_align;       // alignment granularity if section is m
int sh_entsize;     // size of entries if section is an arra
```

Section types include:

- Program contents including code, data, and debugger info.
- Like but no space is allocated in the file itself. Used for BSS data allocated at program load time.
- and Symbol tables, described in more detail later. The table contains all symbols and is intended for the regular linker, while is just the symbols for dynamic linking. (The latter table has to be loaded into memory at runtime.)
- A string table, analogous to the one in a.out files. Unlike a.out files, ELF files can and often do contain separate string tables for separate purposes, e.g. section names, regular symbol names, and dynamic linker symbol names.

- and Relocation information. entries add the relocation value to the base value stored in the code or data, while entries include the base value for relocation in the relocation entries themselves. (For historical reasons, x86 objects use relocation and 68K objects use There are a bunch of relocation types for each architecture, similar to (and derived from) the a.out relocation types.
- and Dynamic linking information and the runtime symbol hash table. There are three flag bits used: which means that the section occupies memory when the program is loaded, which means that the section when loaded is writable, and which means that the section contains executable machine code.

A typical relocatable executable has about a dozen sections. Many of the section names are meaningful to the linker, which looks for the section types it knows about for specific processing, while either discarding or passing through unmodified sections (depending on flag bits) that it doesn't know about.

Sections include:

- which is type PROGBITS with attributes ALLOC+EXECINSTR. It's the equivalent of the a.out text segment.
- which is type PROGBITS with attributes ALLOC+WRITE. It's the equivalent of the a.out data segment.
- which is type with attribute ALLOC. It's read-only data, hence no WRITE.
- which is type NOBITS with attributes ALLOC+WRITE. The BSS section takes no space in the file, hence NOBITS, but is allocated at runtime, hence ALLOC.
- and each which is type REL or RELA. The relocation information for the corresponding text or data section.
- and each type PROGBITS with attributes ALLOC+EXECINSTR. These are similar to but are code to be executed when the program starts up or terminates, respectively. C and Fortran don't need these, but they're essential for C++ which has global data with ex-

ecutable initializers and finalizers.

- and types SYMTAB and DYNSYM respectively, regular and dynamic linker symbol tables. The dynamic linker symbol table is ALLOC set, since it's loaded at runtime.
- and both type STRTAB, a table of name strings, for a symbol table or the section names for the section table. The section, the strings for the dynamic linker symbol table, has ALLOC set since it's loaded at runtime. There are also some specialized sections like and the Global Offset Table and Procedure Linkage Table used for dynamic linking (covered in Chapter 10), which contains symbols for the debugger, which contains mappings from source line numbers to object code locations again for the debugger, and which contains documentation strings, usually version control version numbers.

An unusual section type is which contains the name of a program to use as an interpreter. If this section is present, rather than running the program directly, the system runs the interpreter and passes it the ELF file as an argument. Unix has for many years had self-running interpreted text files, using

```
#! /name/of/interpreter
```

as the first line of the file. ELF extends this facility to interpreters which run non-text programs. In practice this is used to call the run-time loader to load the program and map in any required shared libraries.

The ELF symbol table is similar to the a.out symbol table. It consists of an array of entries, Figure 12.

Figure 12: ELF symbol table

```
int name;// position of name string in string table
int value;// symbol value, section relative in reloc,
     // absolute in executable
```

```
int size;// object or function size
char type:4;// data object, function, section, or special ca
char bind:4;// local, global, or weak
char other;// spare
short sect;// section number, ABS, COMMON or UNDEF
```

The a.out symbol entry is fleshed out with a few more fields. The size field tells how large a data object is (particularly for undefined BSS, the common block hack again.) A symbol's binding can be local, just visible in this module, global, visible everywhere, or weak. A weak symbol is a half-hearted global symbol: if a definition is available for an undefined weak symbol, the linker will use it, but if not the value defaults to zero.

The symbol's type is normally data or function. There is a section symbol defined for each section, usually with the same name as the section itself, for the benefit of relocation entries. (ELF relocation entries are all relative to symbols, so a section symbol is necessary to indicate that an item is relocated relative to one of the sections in the file.) A file entry is a pseudo-symbol containing the name of the source file.

The section number is the section relative to which the symbol is defined, e.g., function entry points are defined relative to Three special pseudo-sections also appear, UNDEF for undefined symbols, ABS for non-relocatable absolute symbols, and COMMON for common blocks not yet allocated. (The value of a COMMON symbol gives the required alignment granularity, and the size gives the minimum size. Once allocated by the linker, COMMON symbols move into the section.)

A typical complete ELF file, Figure 13, contains quite a few sections for code, data, relocation information, linker symbols, and debugger symbols. If the file is a C++ program, it will probably also contain and sections as well.

Figure 13: Sample relocatable ELF file

ELF header

.text
.data
.rodata
.bss
.sym
.rel.text
.rel.data
.rel.rodata
.line
.debug
.strtab
(section table, not considered to be a section)

ELF executable files

An ELF executable file has the same general format as a relocatable ELF, but the data are arranged so that the file can be mapped into memory and run. The file contains a program header following the ELF header which defines the segments to be mapped. The program header, Figure 14, is an array of segment descriptions.

Figure 14: ELF program header

```
int type; // loadable code or data, dynamic linking info, et
int offset;          // file offset of segment
int virtaddr;        // virtual address to map segment
int physaddr;        // physical address, not used
int filesize;        // size of segment in file
int memsize;         // size of segment in memory (bigger if
int flags;           // Read, Write, Execute bits
int align;           // required alignment, invariably hardwa
```

An executable usually has only a handful of segments, a read-only one for the code and read-only data, and a read-write one for read/write data. All of the loadable sections are packed into the appropriate segments so the system can map the file with one or two operations.

ELF files extend the ‘‘header in the address space’’ trick used in QMAGIC a.out files to make the executable files as compact as possible at the cost of some slop in the address space. A segment can start and end at arbitrary file offsets, but the virtual starting address for the segment must have the same low bits modulo the alignment as the starting offset in the file, i.e, must start in the same offset on a page. The system maps in the entire range from the page where the segment starts to the page where the segment ends, even if the segment logicall only occupies part of the first and last pages mapped. Figure 15 shows a typical segment arrangement.

Figure 15: ELF loadable segments

	File offset	Load address	Type
ELF header	0	0x8000000	
Program header		0x40	0x8000040
Read only text	0x100	0x8000100	LOAD, Read/Execute
(size 0x4500)		0x80045ff	
Read/write data	0x4600	0x8005600	LOAD, Read/Write/Execute
(file size 0x2200, memory size 0x3500)			
non-loadable info and optional section headers			

The mapped text segment consists of the ELF header, program header, and read-only text, since the ELF and program headers are in the same page as the beginning of the text. The read/write but the data segment in the file starts immediately after the text segment. The page from the file is mapped both read-only as the last page of the text segment in memory and copy-on-write as the first page of the data segment. In this example, if a

computer has 4K pages, and in an executable file the text ends at 0x80045ff, then the data starts at 0x8005600. The file page is mapped into the last page of the text segment at location 0x8004000 where the first 0x600 bytes contain the text from 0x8004000-0x80045ff, and into the data segment at 0x8005000 where the rest of the page contain the initial contents of data from 0x8005600-0x80056ff. (As a special case, if the text happens to end on a page boundary, the data just starts on the next page and there's no double mapping.)

The BSS section again is logically continuous with the end of the read write sections in the data segment, in this case 0x1300 bytes, the difference between the file size and the memory size. The last page of the data segment is mapped in from the file, but as soon as the operating system starts to zero the BSS segment, the copy-on-write system makes a private copy of the page.

If the file contains or sections, those sections are part of the read only text segment, and the linker inserts code at the entry point to call the section code before it calls the main program, and the section code after the main program returns.

An ELF shared object contains all the baggage of a relocatable and an executable file. It has the program header table at the beginning, followed by the sections in the loadable segments, including dynamic linking information. Following sections comprising the loadable segments are the relocatable symbol table and other information that the linker needs while creating executable programs that refer to the shared object, with the section table at the end.

ELF summary

ELF is a moderately complex format, but it serves its purposes reasonably well. It's a flexible enough relocatable format to support C++, while being an efficient executable format for a virtual memory system with dynamic linking. It also permits cross-compilation and cross-linking from one platform to another, with enough information in each ELF file to identify the target architecture and byte order.

Microsoft Portable Executable format

Microsoft's Windows NT suffers from an extremely mixed heritage including earlier versions of MS-DOS and Windows, Digital's VAX VMS (on which many of the programmers had worked), and Unix System V (on which many of the rest of the programmers had worked.) NT was apparently bootstrapped in part on a 32 bit version of UNIX that used COFF executables, a file format that Unix versions used after a.out but before ELF, and as a result all 32 bit Windows system use a Portable Executable format derived from COFF. We'll take a look at PE and, where it differs from PE, Microsoft's version of COFF.

The PE format is made somewhat more complex by Windows' convoluted history. Early versions of Windows ran in real mode on 8086 processors, then 16 bit segmented protected mode, and now 32 bit unsegmented protected mode. Even the most recent versions of Windows on Intel architectures can still run 286 executables, and a significant amount of 16 bit code remains in Windows 95 and 98. The Windows GUI developed in parallel with the Windows system kernel (unlike Unix where they were completely separate efforts), so there's a much less clear separation between system functions and GIU functions. Finally, Windows developed in an extremely underpowered environment with slow processors, limited RAM, and originally without hardware paging, so there was always an emphasis on shared libraries to save memory, and ad-hoc tricks to improve performance. Every windows executable uses *resources*, a general term that refers to objects such as cursors, icons, bitmaps, menus, and fonts that are shared between the program and the GUI. A PE file contains a resource directory for all of the resources the program uses.

Modern PE executable files are intended for a paged environment, so a PE file is can usually be mapped directly into memory and run, much like an ELF executable. PE's can be either EXE programs or DLL shared libraries (known as dynamic link libraries). The format of the two is the same except that a DLL contains a list of exported functions that can be called by EXEs or other DLLs. Each file contains a set of chunks analogous to ELF segments known variously as sections, segments, and objects. We call them sections here.

A PE file, Figure 15, starts with a small DOS .EXE file which prints out something like "This program needs Microsoft Windows." (Microsoft's dedication to certain kinds of backward compatibility is impressive.) An otherwise unused field at the end of the EXE header points to the PE file header, which in turn points to a section table, which describes the various sections of the file. In a COFF object file, the PE header is at the beginning of the file.

Figure 15: Microsoft PE file

DOS header
DOS program stub
PE header
Section table
Mappable sections (pointed to from section table)
COFF line numbers, symbols, debug info (optional in PE File)

The rather large PE header, Figure 16 describes the contents of the file, with the most important values being the number of entries in the section table, and in the "optional" header, shortcut pointers to the most commonly used file sections. Addresses are all kept as offsets from the place in memory that the program is loaded, also called Relative Virtual Addresses or RVAs.

Figure 16: PE header

```
char signature[4] = "PE\0\0"; // magic number, also shows by
short cputype;      // required CPU, 0x14C for 80386, etc.
short nsect;        // number of sections
int timestamp;      // creation time or zero
int symtaboff;      // file offset of symbol table in COFF o
int nsym; // # entries in COFF symbol table or zero
```

```
short ntsize;       // size of the following optional header
short flags;        // 02 = executable, 0x200 = nonrelocatab
                    //  0x2000 = DLL rather than EXE

// following is optional header that follows PE header,
// not present in COFF objects

// header can be very long, not all fields listed here
int entryaddr;      // RVA of entry point
int imagebase;      // virtual address to map beginning of f
int secalign;       // section alignment, typically 4096, or
int filealign; // file page alignment, typically 512
short osver[2];     // major/minor required OS version
short userver[2]; // major/minor version number of this file
short subsver[2]; // major/minor subsystem version
int imagesize;      // total size of mappable image, rounded
int hdrsize;        // total size of headers up through sect
int checksum;       // nominally file checksum, actually zer
short subsys;       // required subsystem: 1 = native, 2 = W
          // 3 = Windows non-GUI, 5 = OS/2, 7 = POSIX
short dllflags;     // when to call initialization routine (
          // 1 = process start, 2 = process end, 4 = thread
int stacksize;      // size to reserve for stack
int stackalc;       // size to allocate initially for stack
int heapsize;       // size to reserve for heap
int heapalc;        // size to allocate initially for heap
int ndir; // number of entries in following image data direc

// following are entries in image data directory, addresses
// relative to imagebase, tables are described in text
int exporttab;      // RVA of export table .edata
int exportsize;     // size of export table
int importtab;      // RVA of import table .idata
int importsize;     // size of import table
int restab;         // RVA of resource table .rsrc
int ressize;        // size of resource table
```

```
int exceptab;       // RVA of exception table
int excepsize;      // size of exception table
int sectab;         // RVA of security table
int secsize;        // size of security table
int reloctab;       // RVA of fixup (relocation) entries .re
int relocsize;      // size of fixup (relocation) entries
int debugtab;       // RVA of debug table .rdata
int debugsize;      // size of debug table .rdata
int imagedesc;      // RVA of image description string
int imagedsize;     // size of image description string
int machspec;       // RVA of machine specific data
int machsize;       // size of machine specific data
```

Each PE file is created in a way that makes it straightforward for the system loader to map it into memory. Each section is physically aligned on a disk block boundary or greater (the filealign value), and logically aligned on a memory page boundary (4096 on the x86.) The linker creates a PE file for a specific target address at which the file will be mapped (imagebase). If a chunk of address space at that address is available, no load-time fixups are available. Unfortunately, different Windows implementations weren't able to agree on a consistent layout of the address space, so in some cases the target addresses aren't available so the loader has to map the file somewhere else, in which case relocation fixups in the .reloc section tell the loader what to change. (Note that this means the pages after fixups will be different from the version in the file, so many of the benefits of paging directly from the executable are lost.) Shared DLL libraries also are subject to relocation, since the address at which a DLL is mapped depends on what's already occupying the address space.

Following the PE header is the section table, an array of entries like Figure 17.

Figure 17: Section table

```
// array of entries
char name[8];       // section name in ASCII
int vsize;          // size mapped into memory
int addr; // memory address relative to image base
int psize;          // physical size, may be smaller than vs
int offset;         // file offset
          // next four entries exist in COFF, are zero in PE
int relocpos;       // offset of relocation entries (COFF on
int linenopos;      // offset of line number entries (COFF o
short nreloc;       // number of relocation entries (COFF on
short nlineno;      // number of line number entries (COFF o
int flags;          // 0x20 = text, 0x40 = data, 0x80 = bss,
          // 0x800 = don't link, 0x10000000 = shared,
          // 0x20000000 = execute, 0x40000000 = read, 0x8000
```

Each section has both a file address and size (offset and psize) and a memory address and size (addr and vsize) which aren't necessarily the same. The reason is that the CPU's page size is often larger than the disk's block size, typically 4K pages and 512 or 1024 byte disk blocks, and a section that ends in the middle of a page need not have blocks for the rest of the page allocated, saving small amounts of disk space. (Recently, PE files have tended to use the same file page size as CPU page size, making the job of file mapping easier.) Each section is marked with the hardware permissions appropriate for the pages, e.g. read+execute for code and read+write for data.

PE special sections

A PE file includes .text, .data, and .bss files like a Unix executable (usually under those names, in fact) as well as a lot of Windows-specific sections.

- *Exports*: A list of the symbols defined in this module and visible to other modules. EXE files typically export no symbols (or maybe one or two for debugging.) DLLs export symbols for all the routines that they provide. In keeping with Windows space saving tradition, some DLLs name exported entry points using small integers

called export ordinals as well as names. The exports section contains an array of the RVAs of the exported symbol, It also contains two parallel arrays of the name of the symbol (as the RVA of an ASCII string), and the export ordinal for the symbol, sorted by string name. To look up a symbol by name, perform a binary search in the string name table, then find the entry in the ordinal table in the position corresponding to the found name, and use that ordinal to index the array of RVAs. (This is arguably faster than iterating over an array of three-word entries.)

- *Imports*: The imports table lists all of the symbols that need to be resolved at load time from DLLs. The linker predetermines which symbols will be found in which DLLs, so the imports table starts with an import directory, consisting of one entry per referenced DLL. Each directory entry contains the name of the DLL, and parallel arrays one identifying the required symbols, and the other pointing to the place in the image to store the symbol value. The entries in the first value can be either an ordinal (if the high bit is set), or a pointer to a name string preceded by a guess at the ordinal to speed up the search. The second array just lists the one place in the image to store the symbol's value; the linker will already have adjusted all calls to that symbol to call indirectly via that location.

- *Resources*: The resource table is organized as a tree. The structure supports arbitrarily deep trees, but in practice the tree is three levels, resource type, name, and language. (Language here means a natural language, this permits customizing executables for speakers of languages other than English.) Like exported symbols, resources can have both names and numbers. A typical resource might be type DIALOG (Dialog box), name ABOUT (the About This Program box), language English. Unlike symbols which have ASCII names, resources can have Unicode names to support non-English languages. The actual resources are chunks of binary data, with the format of the resource depending on the resource type.

- *Thread Local Storage*: Windows supports multiple threads of execution per process. Each thread can have its own private storage, Thread Local Storage or TLS. This section points to a chunk of the image used to initialize TLS when a thread starts, and also contains pointers to initialization routines to call when each thread starts.
- *Fixups*: If the executable is moved, it is moved as a unit so all fixups have the same value, the difference between the actual load address and the target address. The fixup table, if present, contains an array of fixup blocks, each containing the fixups for one 4K page of the mapped executable. (Executables with no fixup table can only be loaded at the target address.) Each fixup block contains the base RVA of the page, the number of fixups, and an array of 16 bit fixup entries. Each entry contains in the low 12 bits the offset in the block that needs to be relocated, and in the high 4 bits the fixup type, e.g., add 32 bit value, adjust high 16 bits or low 16 bits (for MIPS architecture). This block-by-block scheme saves considerable space in the relocation table, since each entry can be squeezed to two bytes rather than the 8 or 12 bytes the ELF equivalent takes.

Running a PE executable

Starting a PE executable process is a relatively straightforward procedure, considering how complex the PE format is.

- Read in the first page of the file with the DOS header, PE header, and section headers.
- Determine whether the target area of the address space is available, if not allocate another area.
- Using the information in the section headers, map all of the sections of the file to the appropriate place in the allocated address space.
- If the file is not loaded into its target address, apply fixups.

- Go through the list of DLLs in the imports section and load any that aren't already loaded. (This process may be recursive.)
- Resolve all the imported symbols in the imports section.
- Create the initial stack and heap using values from the PE header.
- Create the initial thread and start the process.

PE and COFF

The detailed format of a Windows COFF relocatable object file is similar to that of a PE, but the structure is more similar to that for a relocatable ELF file. COFF files don't have the DOS header nor the optional header following the PE header. Each code or data section also carries along sections with relocation, and line numbers. (The line numbers in an EXE file, if any, are collected in in a debug section not handled by the system loader. COFF objects have section-relative relocations (like ELF files), rather than RVA relative relocations, and invariably contain a symbol table with the symbols needed. COFF files from language compilers typically do not contain any resources, rather, the resources are in a separate object file created by a specialized resource compiler.

COFF files can also have several other section types not used in PE. The most notable is the .drective section which contains text command strings for the linker. Compilers usually use .drective to tell the linker to search the appropriate language-specific libraries. (This mixture of commands and object code goes way back; IBM linkers accepted mixed card decks of commands and object files in the early 1960s.)

PE summary

The PE file format is somewhat overcomplex due to all the historical baggage that Windows carries along. It also involves some extra levels of complication intended to speed up program loading on small systems, but of dubious effectiveness on modern 32 bit systems. In fairness, the earlier NE format for 16 bit segmented executables was even more complicated, and PE is a definite improvement.

Intel/Microsoft OMF files

The final format we look at in this chapter is one of the oldest formats still in use, the Intel Object Module Format. Intel originally defined OMF in the late 1970s for the 8086. Over the years a variety of vendors, including Microsoft, IBM, and Phar Lap (who wrote a very widely used set of 32 bit extension tools for DOS), defined their own extensions. The current Intel OMF is the union of the original spec and most of the extensions, minus a few extensions that either collided with other extensions or were never used.

All of the formats we've seen so far are intended for environments with random access disks and enough RAM to do compiler and linker processing in straightforward ways. OMF dates from the early days of microprocessor development when memories were tiny and storage was often punched paper tapes. As a result, OMF divides the object file into a series of short records, Figure 18. Each record contains a type byte, a two-byte length, the contents, and a checksum byte that makes the byte-wise sum of the entire record zero. (Paper tape equipment had no built-in error detection, and errors due to dust or sticky parts were not rare.) OMF files are designed so that a linker on a machine without mass storage can do its job with a minimum number of passes over the files. Usually 1 1/2 passes do the trick, a half pass to find the symbol names which are placed near the front of each file, and then a full pass to do the linking and produce the output.

Figure 18: OMF record format

picture of
-- type byte
-- two-byte length
-- variable length data
-- checksum byte

OMF is greatly complicated by the need to deal with the 8086 segmented architecture. One of the major goal of an OMF linker is to pack code and data into a minimum number of segments and segment groups. Every piece of code or data in an OMF object is assigned to a segment, and each segment in turn can be assigned to a segment group or segment class. (A group must be small enough to be addressed by a single segment value, a class can be any size, so groups are used for both addressing and storage management, while classes are just for storage management.) Code can reference segments and groups by name, and can also reference code within a segment relative to the base of the segment or the base of the group.

OMF also contains some support for overlay linking, although no OMF linker I know of has ever supported it, taking overlay instructions instead from a separate directive file.

OMF records

OMF currently defines at least 40 record types, too many to enumerate here, so we'll look at a simple OMF file. (The complete spec is in the Intel TIS documents.)

OMF uses several coding techniques to make records as short as possible. All name strings are variable length, stored as a length byte followed by characters. A null name (valid in some contexts) is a single zero byte. Rather than refer to segments, symbols, groups, etc. by name, an OMF module lists each name once in an LNAMES record and subsequently uses a index into the list of names to define the names of segments, groups, and symbols. The first name is 1, the second 2, and so forth through the entire set of names no matter how many LNAMES records they might have taken. (This saves a small amount of space in the not uncommon case that a segment and an external symbol have the same name since the definitions can refer to the same string.) Indexes in the range 0 through 0x7f are stored as one byte. Indexes from 0x80 through 0x7fff are stored as two bytes, with the high bit in the first byte indicating a two-byte sequence. Oddly, the low 7 bits of the first byte are the high 7 bits of the value and the second byte is the low 8 bits of the value, the opposite of the native Intel order. Segments, groups, and external symbols are also referred to by index, with separate index sequences for each. For example, assume a

module lists the names DGROUP, CODE, and DATA, defining name indexes 1, 2, and 3. Then the module defines two segments called CODE and DATA, referring to names 2 and 3. Since CODE is the first segment defined, it will be segment index 1 and DATA will be segment index 2.

The original OMF format was defined for the 16 bit Intel architecture. For 32 bit programs, there are new OMF types defined for the record types where the address size matters. All of the 16 bit record types happened to have even numerical codes, so the corresponding 32 bit record types have the odd code one greater than the 16 bit type.

Details of an OMF file

Figure 19 lists the records in a simple OMF file.

Figure 19: Typical OMF record sequence

THEADR program name
COMENT flags and options
LNAMES list of segment, group, and class names
SEGDEF segment (one record per segment)
GRPDEF group (one record per group)
PUBDEF global symbols
EXTDEF undefined external symbols (one per symbol)
COMDEF common blocks
COMENT end of pass1 info
LEDATA chunk of code or data (multiple)
LIDATA chunk of repeated data (multiple)
FIXUPP relocations and external ref fixups, each following the LEDATA or LIDATA to which it refers
MODEND end of module

The file starts with a THEADR record that marks the start of the module and gives the name of the module's source file as a string. (If this module were part of a library, it would start with a similar LHEADR record.)

The second record is a badly misnamed COMENT record which contains configuration information for the linker. Each COMENT record contains some flag bits saying whether to keep the comment when linked, a type byte, and the comment text. Some comment types are indeed comments, e.g., the compiler version number or a copyright notice, but several of them give essential linker info such as the memory model to use (tiny through large), the name of a library to search after processing this file, definitions of weak external symbols, and a grab-bag of other types of data that vendors shoe-horned into the OMF format.

Next comes a series of LNAMES records that list all of the names used in this module for segments, groups, classes, and overlays. As noted above, the all the names in all LNAMES are logically considered an array with the index of the first name being 1.

After the LNAMES record come SEGDEF records, one for each segment defined in the module. The SEGDEF includes a indexes for the name of the segment, and the class and overlay if any it belongs to. Also included are the segment's attributes including its alignment requirements and rules for combining it with same-name segments in other modules, and its length.

Next come GRPDEF records, if any, defining the groups in the module. Each GRPDEF has the index for the group name and the indices for the segments in the group.

PUBDEF records define "public" symbols visible to other modules. Each PUBDEF defines one or more symbols within a single group or segment. The record includes the index of the segment or group and for each symbol, the symbol's offset within the segment or group, its name, and a one-byte compiler-specific type field.

EXTDEF records define undefined external symbols. Each record contains the name of one symbol and a byte or two of debugger symbol type. COMDEF records define common blocks, and are similar to EXTDEF records except that they also define a minimum size for the symbol. All of the EXTDEF and COMDEF symbols in the module are logically an array, so fixups can refer to them by index.

Next comes an optional specialized COMENT record that marks the end of pass 1 data. It tells the linker that it can skip the rest of the file in the first pass of the linking process. (This probably saved a lot of time when reading files from 8" floppies.)

The rest of the file consists of the actual code and data of the program, intermixed with fixup records containing relocation and external reference information. There are two kinds of data records LEDATA (enumerated) and LIDATA (iterated). LEDATA simply has the segment index and starting offset, followed by the data to store there. LIDATA also starts with the segment and starting offset, but then has a possibly nested set of repeated blocks of data. LIDATA efficiently handles code generated for statements like this Fortran:

```
INTEGER A(20,20) /400*42/
```

A single LIDATA can have a two- or four-byte block containing 42 and repeat it 400 times.

Each LEDATA or LEDATA that needs a fixup must be immediately followed by the FIXUPP records. FIXUPP is by far the most complicated record type. Each fixup requires three items: first the target, the address being referenced, second the frame, the position in a segment or group relative to which the address is calculated, and third the location to be fixed up. Since it's very common to refer to a single frame in many fixups and somewhat common to refer to a single target in many fixups, OMF defines fixup *threads*, two-bit codes used as shorthands for frames or targets, so at any point there can be up to four frames and four targets with thread numbers defined. Each thread number can be redefined as often as needed. For example, if a module includes a data group, that group is usually used as the frame for nearly every data reference in the module, so defining a thread number for the base address of that group saves a great deal of space. In practice a GRPDEF record is almost invariably followed by a FIXUPP record defining a frame thread for that group.

Each FIXUPP record is a sequence of subrecords, with each subrecord either defining a thread or a fixup. A thread definition subrecord has flag bits saying whether it's defining a frame or target thread. A target thread

definition contains the thread number, the kind of reference (segment relative, group relative, external relative), the index of the base segment, group or symbol, and optionally a base offset. A frame thread definition includes the thread number, the kind of reference (all the kinds for target definition plus two common special cases, same segment as the location and same segment as the target.)

Once the threads are defined, a fixup subrecord is relatively simple. It contains the location to fix up, a code specifying the type of fixup (16 bit offset, 16 bit segment, full segment:offset, 8 bit relative, etc.), and the frame and target. The frame and target can either refer to previously defined threads or be specified in place.

After the LEDATA, LIDATA, and FIXUPP records, the end of the module is marked by a MODEND record, which can optionally specify the entry point if the module is the main routine in a program.

A real OMF file would contain more record types for local symbols, line numbers, and other debugger info, and in a Windows environment also info to create the imports and exports sections in a target PE file, but the structure of the module doesn't change. The order of records is quite flexible, particularly if there's no end of pass 1 marker. The only hard and fast rules are that THEADER and MODEND must come first and last, FIXUPPs must immediately follow the LEDATA and LIDATA to which they refer, and no intra-module forward references are allowed. In particular, it's permissible to emit records for symbols, segments, and groups as they're defined, so long as they precede other records that refer to them.

Summary of OMF

The OMF format is quite complicated compared to the other formats we've seen. Part of the complication is due to tricks to compress the data, part due to the division of each module into many small records, part due to incremental features added over the years, and part due to the inherent complexity of segmented program addressing. The consistent record format with typed records is a strong point, since it both permits extension in a straightforward way, and permits programs that process OMF files to skip records they don't understand.

Nonetheless, now that even small desktop computers have megabytes of RAM and large disks, the OMF division of the object into many small records has become more trouble than it's worth. The small record stype of object module was very common up through the 1970s, but is now obsolescent.

Comparison of object formats

We've seen six different object formats in this chapter, ranging from the trivial (.COM) to the florid (PE) to the rococo (OMF). Modern object formats such as ELF try to group all of the data of a single type together to make it easier for linkers to process. They also lay out the file with virtual memory considerations in mind, so that the system loader can map the file into the program's address space with as little extra work as possible.

Each object format shows the style of the system for which it was defined. Unix systems have historically kept their internal interfaces simple and well-defined, and the a.out and ELF formats reflect that in their relative simplicity and the lack of special case features. Windows has gone in the other direction, with process management and user interface intertwined, and at least on x86 systems, a great deal of historical baggage from 16 bit segmented versions of Windows. The PE format reflects that.

Bibliography

AT&T, **System V Application Binary Interface**, Prentice-Hall, 1990, ISBN 0-13-877598-2.

AT&T, **System V Application Binary Interface: Motorola 68000 Processor** Family Supplement" , Prentice-Hall, 1990, ISBN 0-13-877663-6.

AT&T, **System V Application Binary Interface: Intel 386 Supplement**, Intel Corp., 1990, Order number 465681-001.

TIS Committee, **Tool Interface Starndard Portable Formats Specification**, Version 1.1, Intel Order Number 241597, http://developer.intel.com/ial/tis/index.htm.

TIS Committee, **Tool Interface Standard Formats Specification for Windows**, Version 1.0, Intel Order Number 241597, http://developer.in-

tel.com/ial/tis/index.htm.

Matt Pietrek, **Windows 95 System Programming Secrets**, IDG Books, 1995, ISBN 1-56884-318-6.

Intel Corp, **8086 Relocatable Object Module Formats**, 1981, Intel Order Number 121748.