A RUN-TIME LINKING ENVIRONMENT
TO ENABLE DYNAMIC COMPILATION

BY

RICHARD EDWIN KUTTER

B.E.E., University of Dayton, 1996

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1998

Urbana, Illinois

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Figure Page

1. INTRODUCTION

1.1. Overview

The process of building executables from object code modules is a fairly simple one that has been around almost as long as the computer itself. Traditionally, executables are constructed by combining object modules and support code into one unit that is stored for later execution. This model for building software worked well in environments where programs were static and where users used only a few programs. Today, programs are becoming more dynamic – code modules that are used in the same program change frequently and can change even during execution. Clearly, the traditional model of executable construction does not support this emerging trend in software.

This thesis serves several purposes. First, it is designed to illustrate and document the object modules and linking process found in the Microsoft Win32 environment. Second, it is a source of documentation for the software developed as part of this thesis work to solve the problem of dynamic software. Finally, it offers some examples for use of the software developed with this thesis and ideas for future work. This thesis is organized as follows: Chapter 1 provides an introduction and motivation for this work. Chapter 2 describes the format of traditional object modules, the Microsoft Common Object File Format and the Microsoft Archive (Library) file format. Chapter 2 also describes the Microsoft DLL API functions central to the operation of the run-time linker. Chapter 3 describes the traditional linking process under the Microsoft Win32 environment. Chapter 4 describes the design and implementation of the run-time linking software developed for this thesis. Chapter 5 provides some examples of using this software for executing user programs and compares the run-time linker with the Microsoft linker. Finally, Chapter 6 discusses conclusions and future work in this area.

1.2. Motivation

The Java environment created by Sun Microsystems is an example of an environment that supports and promotes the use of the dynamic software paradigm. Researchers at the University of Illinois have been developing a system for improving the performance of Java through the translation of Java bytecode to native code. Their first pass was to create traditional, static executables from the bytecode and the Java library code. This thesis work was conceived to allow their research to more fully conform to the Java specifications. Another motivation for creating such a run-time linking environment is to develop an understanding of what will be necessary for the creation of a run-time linking environment and expose some of the issues surrounding such an environment.

1.3. Related Work

Little is published on the precise mechanics of linkers or dynamic linking for dynamic software. Most of the information for this work has come from the Java Virtual Machine specifications, the Microsoft COFF format specifications, and from experimentation [1], [2]. Most of the current research into Java compilers create static executables, much like the original work of the IMPACT group [3], [4], [5].

Numerous papers have been published on the topic of dynamic linking for mobile code. Franz provides an overview of dynamic linking and load-time code generation [6]. Unfortunately, the literature on dynamic linking through shared libraries does not provide much detail on implementation [6], [7], [8]. This thesis hopes to address this issue by providing a well-documented implementation of dynamic linking using shared libraries.

2. OBJECT FILE FORMAT

2.1. Overview

Object files are the basic machine code building blocks used to create executable programs. To fully appreciate the operation of the linker, one must first understand the structure of the files the linker operates on. Microsoft's Common Object File Format (COFF) is derived from the "standard" UNIX COFF format. Microsoft has added some additional fields to allow its linker to function with a wider array of programming languages. This chapter describes Microsoft's implementation of the Common Object File Format for object files (not executable files). Section 2.2 describes the COFF header, Section 2.3 describes the section tables of the COFF file, and Sections 2.4 and 2.5 describe the symbol table and string table. Section 2.6 describes the differences between Microsoft COFF and the Binary Format (BF) object files designed specifically for this work. Next, Section 2.7 describes the structure of library files, the Microsoft Archive Format. Finally, Section 2.8 documents a subset of the Microsoft DLL API functions and includes a simple example to illustrate its use.

Figure 2.1 illustrates the layout of a typical Microsoft COFF file. The shaded areas indicate optional portions.

2.2. COFF Header

Attributes stored in the COFF header must be known to the linker prior to acting on the remainder of the file. The COFF header indicates the type of machine the object file was created for, the number of sections, the date and time of file creation, the offset to the symbol table, the number of symbols, and a set of flags as shown in Table 2.1.

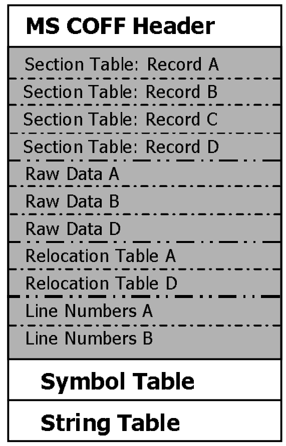

Figure 2.1 COFF Object File Layout

Table 2.1 Fields in COFF Header

Field Name	Size (in bytes)	Offset (from start of file, in bytes)	Description
Machine Type	2	0	Numerical machine type ID (Section 2.2.1)
Number of Sections	2	2	Indicates size of section table
Time/Date Stamp	4	4	UNIX-style representation of time/date (time_t)
Ptr to Symbol Table	4	8	Offset within file to symbol table
Number of Symbols	4	12	Number of symbols in symbol table
Optional Header Size	2	16	Size of optional header (only for executables)
Characteristics	2	18	See Section 2.2.2

2.2.1. Machine types

The 2-byte fields shown in Table 2.2 represent machine (CPU) type [9]. An executable file of a specific machine type can only be run on the specified machine or a system emulating it.

2.2.2. COFF characteristics flags

The characteristics flags field contains flags that describe attributes of the object file. Currently defined flags are shown in Table 2.3. These definitions are from `winnt.h` and the Microsoft PE and COFF Specification, version 4.1 [2], [9].

Table 2.2 Machine Types

Constant (WINNT.H)	Value	Description
IMAGE_FILE_MACHINE_UNKNOWN	0x0000	Assumed to be useful for all types
IMAGE_FILE_MACHINE_I386	0x014C	Intel 386 and later
IMAGE_FILE_MACHINE_R3000	0x0162	MIPS R3000 little-endian, 0x160 big
IMAGE_FILE_MACHINE_R4000	0x0166	MIPS R4000 little-endian
IMAGE_FILE_MACHINE_R10000	0x0168	MIPS R10000 little-endian
IMAGE_FILE_MACHINE_ALPHA	0x0184	Alpha AXP
IMAGE_FILE_MACHINE_POWERPC	0x01F0	PowerPC little-endian
IMAGE_FILE_MACHINE_M68K	0x0268	Motorola M68000 series
IMAGE_FILE_MACHINE_PARISC	0x0290	HP PA RISC

Table 2.3 Characteristics Flags

Flag	Value	Description
IMAGE_FILE_RELOCS_STRIPPED	0x0001	Relocation info stripped from file
IMAGE_FILE_EXECUTABLE_IMAGE	0x0002	File is executable (i.e. no unresolved external references)
IMAGE_FILE_LINE_NUMS_STRIPPED	0x0004	Line numbers stripped from file
IMAGE_FILE_LOCAL_SYMS_STRIPPED	0x0008	Local symbols stripped from file
IMAGE_FILE_AGGRESIVE_WS_TRIM	0x0010	Aggressively trim working set
IMAGE_FILE_UPDATE_OBJECT	0x0020	Reserved for future use
IMAGE_FILE_BYTES_REVERSED_LO	0x0080	Little endian: LSB precedes MSB
IMAGE_FILE_32BIT_MACHINE	0x0100	32-bit word machine
IMAGE_FILE_DEBUG_STRIPPED	0x0200	Debugging info stripped from file and placed in a .DBG file
IMAGE_FILE_REMOVABLE_RUN_FROM_SWAP	0x0400	If Image is on removable media, copy and run from the swap file
IMAGE_FILE_NET_RUN_FROM_SWAP	0x0800	If Image is on Net, copy and run from the swap file
IMAGE_FILE_SYSTEM	0x1000	System file
IMAGE_FILE_DLL	0x2000	File is a DLL
IMAGE_FILE_UP_SYSTEM_ONLY	0x4000	File should only be run on an “UP” machine – as described in the documentation
IMAGE_FILE_BYTES_REVERSED_HI	0x8000	Big endian

2.3. Section Table

Each section is first described by a section table (a.k.a. section header), which holds information about that section. The section table holds such information as the section name, size, pointers to data, relocation information, line number information, and section flags. Table 2.4 illustrates the structure of data in the section table.

Table 2.4 Fields in Section Table

Field Name	Size	Offset	Description
Section Name	8	0	Up to 8-byte, null-padded ACSII string, no null if exactly 8 bytes in length. For longer names, the field is a '/' followed by an ACSII representation of a decimal offset into the string table (Section 2.5).
Virtual Size	4	8	Total size of section after being loaded into memory, zero for object files.
Offset	4	12	For object files, this is the address of the first byte before relocation – typically set to zero. Otherwise, it is an arbitrary value that is subtracted from offsets during relocation (according to documentation). Has alternate definition for executable files.
Raw Data Size	4	16	Size of section or size of initialized data on disk.
Ptr to Raw Data	4	20	File pointer to section's first page within the COFF file. Should be aligned on a four-byte boundary for best performance in object files.
Ptr to Relocs	4	24	File pointer to beginning of relocation entries. Set to 0 for executable files.
Ptr to Linenums	4	28	File pointer to beginning of line-number entries for the section.
# of Relocs	2	32	Number of relocation entries for the section.
# of Linenums	2	34	Number of line-number entries for the section.
Section Flags	4	36	Flags describing section characteristics. See Section 2.3.1 for details.

2.3.1. Section flags

The section flags defined in `winnt.h` and the Microsoft PE and COFF Specification describe the characteristics of a section, shown in Table 2.5 [2], [9].

2.3.2. COFF relocation table

COFF relocations instruct the linker how to modify section data when the object file is built into an executable image. The COFF relocation table contains an array of fixed-length records. The section header specifies the position and number of elements in the array. Table 2.6 defines the fields of each fixed-length record in the relocation table.

Table 2.5 Section Flags

Flag	Value	Description
IMAGE_SCN_TYPE_REG	0x00000000	Reserved
IMAGE_SCN_TYPE_DSECT	0x00000001	Reserved
IMAGE_SCN_TYPE_NOLOAD	0x00000002	Reserved
IMAGE_SCN_TYPE_GROUP	0x00000004	Section is grouped with other sections
IMAGE_SCN_TYPE_NO_PAD	0x00000008	Section should not be padded to next boundary
IMAGE_SCN_TYPE_COPY	0x00000010	Reserved
IMAGE_SCN_CNT_CODE	0x00000020	Section contains executable code
IMAGE_SCN_CNT_INITIALIZED_DATA	0x00000040	Section contains initialized data
IMAGE_SCN_CNT_UNINITIALIZED_DATA	0x00000080	Section contains uninitialized data
IMAGE_SCN_LNK_OTHER	0x00000100	Reserved
IMAGE_SCN_LNK_INFO	0x00000200	Section contains comments or other information – often linker options
IMAGE_SCN_TYPE_OVER	0x00000400	Reserved
IMAGE_SCN_LNK_REMOVE	0x00000800	Section contents will not become part of image
IMAGE_SCN_LNK_COMDAT	0x00001000	Section contents COMDAT
IMAGE_SCN_MEM_PROTECTED	0x00004000	Obsolete
IMAGE_SCN_MEM_FARDATA	0x00008000	Reserved
IMAGE_SCN_MEM_SYSHEAP	0x00010000	Obsolete
IMAGE_SCN_MEM_PURGEABLE	0x00020000	Purgeable
IMAGE_SCN_MEM_16BIT	0x00020000	Not defined in documentation
IMAGE_SCN_MEM_LOCKED	0x00040000	Not defined in documentation
IMAGE_SCN_MEM_PRELOAD	0x00080000	Not defined in documentation
IMAGE_SCN_ALIGN_1BYTES	0x00100000	1 byte alignment
IMAGE_SCN_ALIGN_2BYTES	0x00200000	2 byte alignment
IMAGE_SCN_ALIGN_4BYTES	0x00300000	4 byte alignment
IMAGE_SCN_ALIGN_8BYTES	0x00400000	8 byte alignment
IMAGE_SCN_ALIGN_16BYTES	0x00500000	16 byte alignment (default)
IMAGE_SCN_ALIGN_32BYTES	0x00600000	32 byte alignment
IMAGE_SCN_ALIGN_64BYTES	0x00700000	64 byte alignment
IMAGE_SCN_LNK_NRELOC_OVFL	0x01000000	Section contains extended relocations
IMAGE_SCN_MEM_DISCARDABLE	0x02000000	Section can be discarded
IMAGE_SCN_MEM_NOT_CACHED	0x04000000	Section is not cacheable
IMAGE_SCN_MEM_NOT_PAGED	0x08000000	Section is not pageable
IMAGE_SCN_MEM_SHARED	0x10000000	Section is shareable
IMAGE_SCN_MEM_EXECUTE	0x20000000	Section is executable
IMAGE_SCN_MEM_READ	0x40000000	Section is readable
IMAGE_SCN_MEM_WRITE	0x80000000	Section is writeable

Table 2.6 Fields of COFF Relocation Table

Field	Size	Offset	Description
Offset	4	0	Offset from the beginning of the current section. Used in calculating where the relocation will be applied
Symbol Table Index	4	4	A zero-based index into the symbol table (0, 1, 2, . . .). The symbol table yields the address of the item used in the relocation.
Type	2	8	Indicates the type of relocation to be performed

2.3.3. Relocation types

The type fields of the relocation table indicate the kind of relocation that should be performed. Different relocation types are defined for all supported machine architectures. Table 2.7 lists the relocation types for the Intel 386 architecture [2], [9].

Table 2.7 Relocation Types

Constant	Value	Description
IMAGE_REL_I386_ABSOLUTE	0x0000	Reference is absolute, no relocation is necessary
IMAGE_REL_I386_DIR16	0x0001	Direct 16-bit reference to the symbol's virtual address
IMAGE_REL_I386_REL16	0x0002	PC-relative 16-bit reference to the symbol's virtual address
IMAGE_REL_I386_DIR32	0x0006	Direct 32-bit reference to the symbol's virtual address, plus the Offset field in the Relocation Table (Section 2.3.2)
IMAGE_REL_I386_DIR32NB	0x0007	Direct 32-bit reference to the symbol's virtual address, base virtual address of loaded executable not included
IMAGE_REL_I386_SEG12	0x0009	Direct 16-bit reference to the segment-selector bits of a 32-bit virtual address
IMAGE_REL_I386_SECTION	0x000A	Reference to section address
IMAGE_REL_I386_SECREL	0x000B	Reference to offset from a section address
IMAGE_REL_I386_REL32	0x0014	PC-relative 32-bit reference to the symbols virtual address

2.3.4. COFF line numbers

The COFF line number table is used to relate machine operations to source code lines for analysis tools such as a debugger. The Microsoft COFF format has extended standard COFF line numbers to allow a single section to relate to line numbers in multiple source files. The COFF line number table consists of an array of fixed-length records. The size and location of the table are specified in the section table. The format for each line number record is shown in Table 2.8. Line number records represent either a standard line number entry or point to a function definition in the symbol table, depending on whether the line number field contains a line number or zero, respectively.

Table 2.8 Fields in COFF Line Numbers Table

Field		Size	Offset	Symbol
Type (union)	Symbol Table Index	4	0	When Line Number is 0, this field is a zero-based index into the symbol table
	RVA	4	0	When Line Number is greater than zero, this field is a relative address with respect to the beginning of the section for the executable code that corresponds to the source line indicated in Line Number
Line Number		2	4	When nonzero, specifies a one-based line number, otherwise, it is interpreted as a Symbol Table Index for a function

2.4. Symbol Table

Derived directly from the traditional COFF symbol table, the Microsoft COFF symbol table is used to store section names, source file names, code and data symbols, global variable sizes, and function addresses in the code. The location and size of the symbol table is given in the COFF Header. Like other tables in the object file, the symbol table consists of an array of fixed-length (18 byte) records. Unlike other tables, each record is either a standard type record or an auxiliary record. A standard symbol table record is used to define a symbol or name, and is shown in Table 2.9. Zero or more auxiliary records may follow a standard record and each may be in one of several different formats, depending on the Storage Class field. Auxiliary records are documented in the Microsoft Specification [2].

Table 2.9 Fields in Symbol Table

Field		Size	Offset	Description
Name (union)	Short Name	8	0	Array of 8 characters (bytes), zero padded if less than 8. This is the name of the symbol if the symbol name is 8 bytes long or less
	Zeros	4	0	Set to zero if the symbol name is longer than eight bytes
	Offset	4	4	Offset into the String Table if the symbol name is longer than 8 bytes
Value		4	8	Value associated with the symbol. The interpretation of this field is based on the Section Number and Storage Class. Typically, this field describes the address to be used for relocation
Section Number		2	12	One-based index into the Section Table. Some (< 1) values have special meaning defined in Section 2.4.1
Type		2	14	The Type field is described in Section 2.4.2
Storage Class		1	16	Storage classes are defined in Section 2.4.3
# of Aux Symbols		1	17	Number of auxiliary entries that follow this record

2.4.1. Section number values

Typically, the section number is a one-based index into the Section Table. Being a signed integer, this field may take special values less than one, which are listed in Table 2.10 [2], [9].

Table 2.10 Special Section Number Values

Constant	Value	Description
IMAGE_SYM_UNDEFINED	0	This symbol record is not yet assigned a section. External symbols and global variables use this value in object files
IMAGE_SYM_ABSOLUTE	-1	The symbol has a value, but it is not an address. Automatic variables and register variables fall into this category
IMAGE_SYM_DEBUG	-2	This symbol provides general type or debugging information. This type is most often used for storing the source file name

2.4.2. Type representation

The Type field holds type information in a 2-byte field, where each byte represents separate type information. The least significant byte represents the simple (base) data type, which can be integer, floating-point, etc., and the most significant byte represents complex types such as functions, pointers, arrays, etc. Microsoft tools typically use only the complex type, defining a function as 0x20 and "not a function" as 0x0. Further details are given in the Microsoft PE and COFF Specification [2].

2.4.3. Storage class

The Storage Class field of the symbol table is an unsigned byte that indicates what kind of definition a symbol represents. Table 2.11 lists the most common values held by the Storage Class field and their meanings. The Microsoft PE and COFF Specification contain a more complete listing [2].

Table 2.11 Common Storage Class Definitions

Constant (winnt.h)	Value	Description
IMAGE_SYM_CLASS_EXTERNAL	0x02	Indicates an external symbol. If the section number is 0 (IMAGE_SYM_UNDEFINED), then the Value field specifies the size, otherwise, the Value field specifies the offset within the section
IMAGE_SYM_CLASS_STATIC	0x03	The Value field specifies the offset of the symbol in the section. If the Value is 0, the symbol represents a section name
IMAGE_SYM_CLASS_LABEL	0x06	Code label defined within the module. Value field specifies the offset of the symbol in the section
IMAGE_SYM_CLASS_FILE	0x67	Used for the source file symbol record. An auxiliary record that gives the name of the source file always follows this symbol

2.5. COFF String Table

The last data structure in an object file is the COFF string table, which immediately follows the COFF symbol table. The position of this table is calculated by adding the number of symbols multiplied by the size of a symbol to the address of the symbol table from the COFF header. The string table begins with four bytes that describe the size of the string table, including the size field. The size field is interpreted as an integer on the architecture defined in the COFF header. Variable length null-terminated strings follow the Size field. Pointers in the symbol table and section table reference these strings.

2.6. BF File Format Differences

The BF File format was developed as a smaller, quick-linking version of Microsoft COFF. Each part of the Microsoft COFF object file structure has been altered slightly to decrease file size and complexity. The BF format does not contain any debugging information and partial linking is performed when possible during conversion from Microsoft COFF to BF.[1] Typically, the BF format represents a 200-byte reduction in size over the Microsoft COFF file size.

2.6.1. BF header

The BF Header corresponds roughly to the COFF Header (Section 2.2). The BF Header contains a four-byte magic number and major and minor version numbers in addition to the standard COFF fields. The BF Header does not have the Optional Header Size field and or a Time/Date Stamp field. The BF Header is shown in Table 2.12.

Table 2.12 BF Header

Field	Size	Offset	Description
Magic	4	0	Magic Number = 0xFACEBABE (originally 0xBABEFACE for BF Format, but BABEFACE has already been used)
Minor	2	4	Minor Version: Used for compatibility with Java bytecode format
Major	2	6	Major Version: Used for compatibility with Java bytecode format
Machine	2	8	Machine Type – same as Microsoft COFF
# of Sections	2	10	Number of sections
Ptr to Symbol Table	4	12	File offset to symbol table
# of Symbols	2	16	Number of symbols
Flags	2	18	Reserved

[1] Partial linking is defined to be the process of combining like sections within an object module without creating an executable.

2.6.2. BF section table

The BF Section Table is also loosely based on the Microsoft COFF Section Table (Section 2.3). The Virtual Address and all Line Number fields are removed and an "unused" field is added to the BF Section Table. Since all Microsoft tools use section names less than or equal to eight characters, the Name field is only interpreted as a string, rather than as a union (see Table 2.4 on page 7). Table 2.13 describes the structure of the BF Section Table.

Table 2.13 Fields in BF Section Table

Field	Size	Offset	Description
Name	8	0	Section Name
Raw Size	4	8	Size of Raw Data
Ptr to Raw	4	12	Pointer to Raw Data
Ptr to Reloc	4	16	Pointer to Relocation Table
# of Relocs	2	20	Number of Relocations
Unused	2	22	Used by Linker to Lock Sections
Flags	4	24	Same as Characteristics in COFF Section Table

2.6.3. BF relocation table

The BF Relocation Table has the same structure as the Microsoft COFF Relocation Table (Section 2.3.2). Table 2.14 illustrates the structure of the BF Relocation Table.

Table 2.14 Fields in BF Relocation Table

Field	Size	Offset	Description
Offset	4	0	Offset into current section
Symbol Index	4	4	Zero-based index into Symbol Table
Type	2	8	Value indicating relocation type – same as Microsoft COFF

2.6.4. BF symbol table

The BF Symbol Table is identical to the Microsoft COFF Symbol Table (Section 2.4), except that the Number of Auxiliary Symbols field is replaced with a Flags field. The run-time linker uses the Flags field to mark symbols that point to global variables and uninitialized variables.

2.6.5. BF string table

The BF String Table is identical to the Microsoft COFF String Table described in Section 2.5.

2.6.6. BF file construction

Binary format files were to be created by a Java bytecode to native code translator. For the purpose of testing the run-time linker, BF files are created by converting Microsoft COFF object files. During the conversion process, any partial linking that can be performed is done, auxiliary entries in the Symbol Table are dropped, and line number entries are dropped. The `pe_to_bf` program described in Appendix A performs the described file conversions.

2.7. Microsoft Archive File Format

The archive file format (also known as library file format) is designed to implement an efficient interface with the linker and is the standard method of storing collections of object files. Archive files are identified by an eight-byte signature at the beginning of the file. The rest of the object file consists of a series of sections called archive members. The first and second members are called "linker members." These members list objects available for the linker and each has a separate format. The optional third member is called the longnames member, which consists of a special header, followed by a table of null-terminated ASCII strings. The rest of the archive consists of standard object files. Figure 2.2 illustrates the structure of an archive. The Microsoft LIB program is used to manipulate archive files.

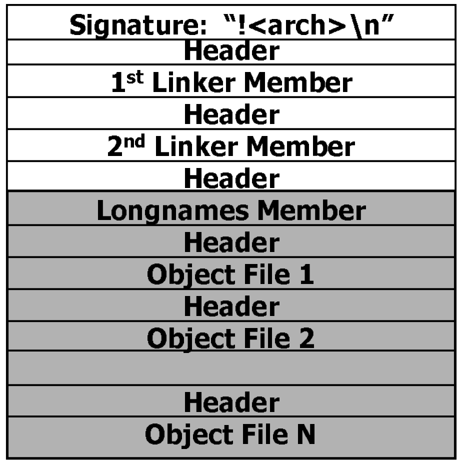

Figure 2.2 Structure of Microsoft COFF Archive

2.7.1. Archive file signature

The archive file signature is inherited from the UNIX COFF magic number for archive files. Utilities expecting an archive file as input can check the file type by reading the archive signature. The signature is the following string, where '\n' is interpreted as a line-feed character: "!<arch>\n."

2.7.2. Archive member headers

An archive member header precedes each member of an archive. The header has a simple structure that interfaces easily with linkers on many platforms. Each field of the header is a left-justified ASCII text string and is padded with spaces to the end of the field. This format is easily converted to the native data type by the C standard library function, *sscanf()*.

Each member header begins on the first even address after the end of the previous archive member. Table 2.15 illustrates the structure of the archive member header.

Table 2.15 Structure of Archive Member Header

Field	Size	Offset	Description
Name	16	0	Name of archive member, terminated with '/'. If the first character is a '/', the name has special interpretation
Date	12	16	Data and time the archive was created: ASCII decimal representation of UNIX time_t field
User ID	6	28	ASCII decimal representation of User ID
Group ID	6	34	ASCII decimal representation of Group ID
Mode	8	40	ASCII octal representation of member's files mode
Size	10	48	ASCII decimal representation of the total size of the archive member, not including the size of the header
End of Header	2	58	The two bytes in the string "'\n" (apostrophe followed by the new-line character)

2.7.3. Interpreting the name field

The name field can be interpreted in a number of ways, depending on its contents. Each of these fields is left-justified and padded with spaces within a field of 16 bytes.

Table 2.16 Name Field Interpretations

Contents	Description
name/	The name of the archive member is given directly
/	This archive member is one of the two linker members (both have the same name).
//	This archive member is the longnames member. If present, it is the third member in an archive (See Section 2.7.6)
/*n*	The name of the archive is located at offset *n* within the longnames member

2.7.4. First linker member

The first linker member is provided for backward compatibility with UNIX archive files. The name of the first linker member is "/". Symbol names from all of the included object files are listed in the first linker member in order of their appearance. Each symbol is associated with an offset that points to the member header of the object file containing the symbol. Using the linker members, the linker can quickly locate needed symbols and extract the object files containing information needed by the linker. The first linker member is also referred to as the archive symbol list or archive symbol table. Table 2.17 illustrates the structure of the first linker member, which follows the header.

Table 2.17 Structure of First Linker Member

Field	Size	Offset	Description
Number of Symbols	4	0	Unsigned long representing number of symbols indexed. Big-endian.
Offsets	4*n	4	Array of file offsets to archive member headers, in which n is the Number of Symbols. Big-endian. Gives the offset in the archive of the header of the member containing the symbol.
String Table	*	*	Series of null-terminated strings that name all of the symbols in the archive. Each string begins immediately after the null character of the previous string.

2.7.5. Second linker member

The second linker member is an enhancement to the UNIX COFF style archive that stores symbols in lexical order for faster searching. The name of the second linker member is also "/". The second linker member follows the header and is illustrated in Table 2.18.

Table 2.18 Structure of Second Linker Member

Field	Size	Offset	Description
Number of Members	4	0	Unsigned long storing the number of members.
Offsets	4*m	4	Array of file offsets to archive member headers, arranged in ascending order. Each is an unsigned long. m = Number of Members.
Number of Symbols	4	*	Unsigned long representing the number of symbols indexed.
Indices	2*n	*	Array of 1-based indices (unsigned short) which map symbol names to archive member offsets. n = Number of Symbols.
String Table	*	*	Same as String Table in First Linker Member.

2.7.6. Longnames member

The longnames member stores names of members that are too long to fit in the Name field of the header (greater than 16 bytes). The longnames member, named "//", is optional. A string table like that in the first and second linker members immediately follows the header. All members following the longnames member contain object modules.

2.8. Microsoft DLL API

This section describes the API interface used by the run-time linker to dynamically load (and potentially unload) DLL libraries. The API functions *LoadLibrary*, *GetProcAddress*, and *FreeLibrary* are described. Then, a simple program demonstrating their use is shown and described.

The run-time linker uses the DLL handling routines to perform run-time linking of code provided in the Microsoft C run-time library and system libraries. This approach to using library code saves memory at the cost of linking speed. Traditional linkers typically link most of the standard C library functions into the executable they produce, wasting valuable memory by maintaining separate copies of code that can be shared by all processes. The Microsoft DLL routines also allow the run-time linker to place responsibility on the operating system to correctly load, link, and unload the library functions that are used by the run-time linker.

2.8.1. LoadLibrary

The *LoadLibrary* API function simply maps the specified executable module (DLL) into the address space of the calling process [10]. When *LoadLibrary* is called, the specified DLL is located by searching the following locations, in sequence:

1. The directory containing the executable for the current process.

2. The current directory.
3. The Windows *system* directory (i.e., `C:\WINDOWS\system32`).
4. The Windows directory (i.e., `C:\WINDOWS`).
5. The directories listed in the PATH environment variable.

If the search succeeds, the system maps the DLL module into the virtual address space of the process and increments the reference count for the DLL. The reference count is used to determine when to unmap the DLL from the executable's address space. If the DLL specified is already mapped into the address space of the process, the reference count is incremented and a handle to the DLL is returned to the calling process. The system automatically loads other executable modules referenced by the specified DLL. Finally, if the current call is the first time the DLL has been loaded, the DLL entry-point function is called and the module is initialized. The handle to the DLL is returned only if the entry-point function exits without an error condition. *LoadLibrary* expects one argument, a null-terminated string representing the name of the executable module. The function returns NULL if an error has occurred.

2.8.2. GetProcAddress

The *GetProcAddress* API function returns the address of the specified exported DLL function [10]. *GetProcAddress* expects two arguments, the handle to the DLL and a null-terminated string representing the name of the exported DLL function. The spelling and case of the function name specified must be identical to that in the exports section of the DLL. If *GetProcAddress* fails, it returns the NULL pointer to the calling program.

2.8.3. FreeLibrary

FreeLibrary decrements the reference count of the loaded DLL module and unmaps the DLL from the process address space if the computed reference count is zero [10]. When the module is unmapped from the address space, the handle to the module is invalidated. The function expects a module handle as its input argument. *FreeLibrary* returns a nonzero value if the function succeeds and zero if an error condition is present.

2.8.4. DLL API example

This section describes a small example program that demonstrates the use of the DLL API functions for dynamic linking. The sample program, `poc.c` (Figure 2.3), dynamically loads and links the *MessageBox* function that places a popup window on the windows desktop with a simple message.

The first line of executable code in `poc.c` uses the *GetModuleHandle* API function to determine if `user32.dll` has been loaded. This is included to show that `user32.dll` is not automatically loaded by `poc.exe`. Next, *LoadLibrary* is used to load `user32.dll` into the address space of `poc.exe`. Then, after checking if *LoadLibrary* succeeded, *GetProcAddress* is used to get the address of the function *MessageBoxA*. After the address of *MessageBoxA* is successfully determined, *MessageBox* is called with the parameters: NULL, "Hey it worked," "Press okay to continue," and zero. The first parameter is the handle to the parent window and is NULL in this case because `poc.exe` is a command line application. The second parameter is the message box text and the third parameter is the window title. The fourth parameter is a set of flags that determines which buttons are displayed and what actions they perform. A zero represents a simple "OK" button that waits for user input before continuing execution. Finally, `user32.dll` is unmapped from the address space of `poc.exe` by the *FreeLibrary* function. Figure 2.4 shows the command line output and Figure 2.5 illustrates the message box that is displayed during execution of `poc.exe`.

```
/* poc.c    Rich Kutter    12 January, 1998         */
/***************************************************/
/*  Proof of concept of run-time dynamic linking    */
/***************************************************/

#include <stdio.h>
#include <windows.h>
#include <winbase.h>

/* Function pointer type that accepts the same arguments as
   MessageBox */
typedef void (*FcnPtr)(int, char *, char *, unsigned int);


void main(void) {
    int i;
    HINSTANCE DLLHandle;
    FcnPtr FcnAddr;
    int fFreeResult, fRunTimeLinkSuccess = FALSE;

    // Try to see if the DLL user32.dll was loaded at load-time
    // If it was, GetModuleHandle will return a handle for user32.dll
    // If not, it returns NULL and the DLL must be mapped into our
    // address space using LoadLibrary

    DLLHandle = GetModuleHandle("user32");
    if (DLLHandle == NULL) {
       printf("GetModuleHandle  failed,  must  use  LoadLibrary  to  load
DLL!\n");

       // Get a handle to the DLL module and load into this programs space.
       DLLHandle = LoadLibrary("user32");
    } else {
       printf("GetModuleHandle worked!\n");
    }

    // If the handle for user32.dll is valid, try to get the address
    // of the function "MessageBoxA".
    if (DLLHandle != NULL) {
       FcnAddr = (FcnPtr) GetProcAddress(DLLHandle, "MessageBoxA");
        // If the function address is valid, call the function.
       if (FcnAddr != NULL)
            // This is the same as calling MessageBox
              (FcnAddr)(NULL,"Hey it worked!","Click OK to continue", 0);
       // Free the DLL module.
       fFreeResult = FreeLibrary(DLLHandle);
        // See if it was actually freed
        if (fFreeResult != NULL) {
           printf("User32.dll was freed with the following code: %d\n",
             fFreeResult);
        }
    }
}
```

Figure 2.3 Source Code for DLL API Example, poc.c

```
C:\test> poc
GetModuleHandle failed, must use LoadLibrary to load DLL!
User32.dll was freed with the following code: 1
C:\test>
```

Figure 2.4 Command line output from poc.exe

Figure 2.5 Dynamically loaded and linked MessageBox window

3. TRADITIONAL LINKING

3.1. Overview

This chapter will help the reader understand how executables are created using a standard linker. The chapter begins with a simple explanation of a linker (Section 3.1) and then explains the operation of a linker in greater depth (Sections 3.2 - 3.4). Section 3.2 explains concatenation, the simplest and most obvious job of the linker. Section 3.3 covers the resolution of external declarations. Finally, Section 3.4 explains the concept of a fixup and how it is performed.

A linker is a program that combines one or more object modules into an executable file. An object module is typically considered to be a Microsoft COFF format object file of the format described in Chapter 2. The most important contents of an object file are the machine code and data. This information is organized into contiguous blocks called sections. Each section is assigned a name: machine code is usually stored in a section called `.text` and initialized data is stored in a section called `.data`. These names are inherited from the UNIX COFF standard and have no special meaning. Table 3.1 lists commonly encountered section names [11].

3.2. Concatenation

Concatenation is the most important (and simple) job of the linker, and is the process of collecting all the object modules and combining all the sections from the object files. The Microsoft linker uses a set of rules to determine how to concatenate and arrange the sections from the various object files in the final executable. The primary rule is that the linker must include all the code and data sections from the specified object files into the executable. Another rule observed by the linker is that each section of the same name is concatenated in the executable. For example, if the three object files specified each had a `.text` section, the resulting executable would have a single `.text` section consisting of the three `.text` sections concatenated in the order they were encountered. Another rule is that the order of sections in the executable is determined by the order in which the sections are encountered by the linker. Figure 3.1 illustrates the concatenation process as determined by the currently defined set of rules. The object files illustrated each have a `.text` and `.data` section in a different location as well as another section that is unique to each object file. Invoking the Microsoft linker, `LINK`, with the arguments `a.obj b.obj c.obj` results in an executable `a.exe`, which is similar to what is shown in Figure 3.1.

Table 3.1 Common Section Names Used by Microsoft Tools

Section Name	Description
.text	Machine code instructions
.data	Initialized data
.bss	Uninitialized data (rarely found in executables – usually merged with .data)
.rdata	Read only data
.edata	The exported function table. Created by the linker – usually found in DLLs
.CRT	Tables of initialization and shutdown pointers used by the runtime library
.CRT$XXX	Initialization and shutdown pointers in object files before concatenation
.idata	Imported function table in an executable
.idata$XX	Portions of an imported function table before concatenation by the linker
.debug$XXX	COFF symbol table information in object file
.reloc	Base relocation section produced by the linker. Not found in object files
.drectve	Object file section containing linker directives – no included in executable

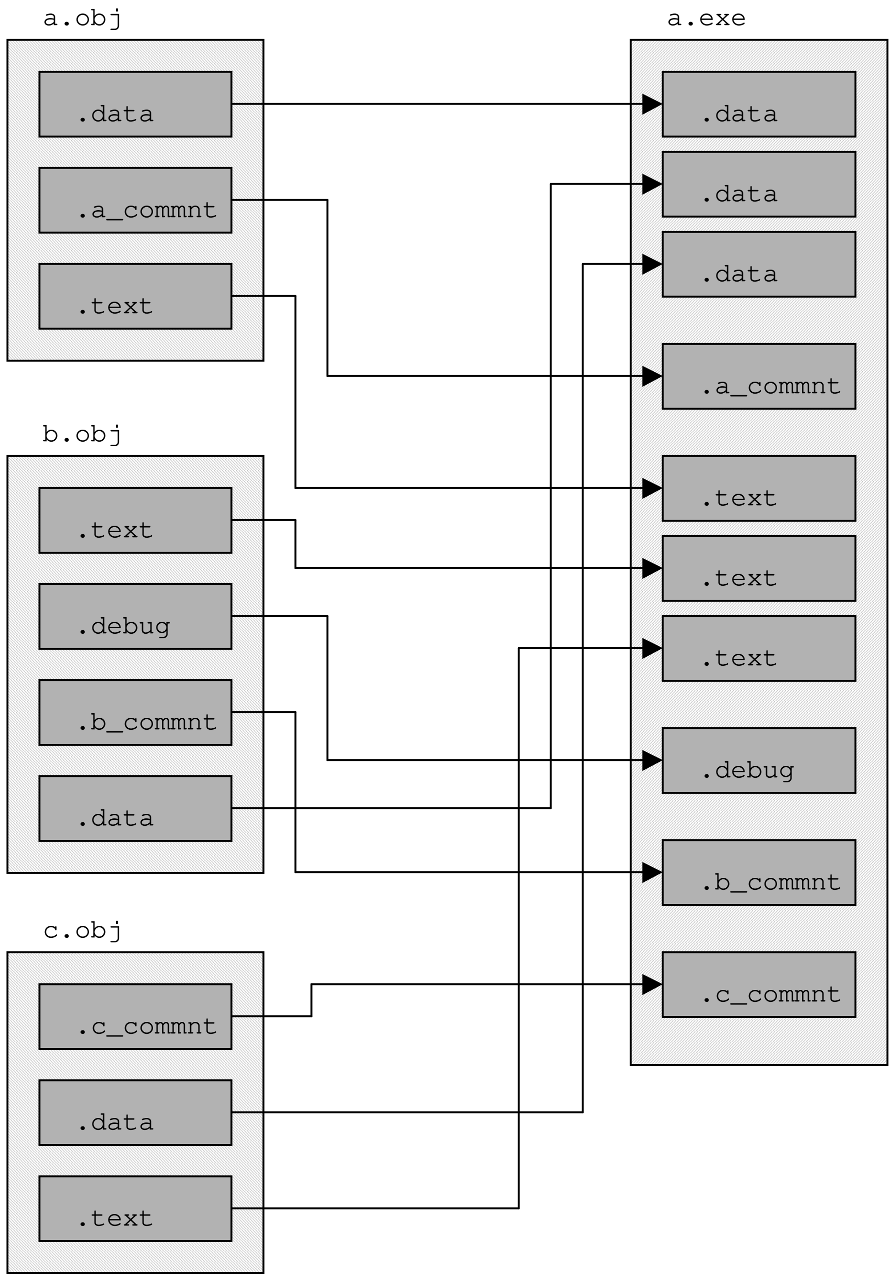

Figure 3.1 Simple Concatenation Example

The rules given thus far are only half the picture. After applying the rules listed above, the Microsoft Win32 Linker has a number of special cases to consider. The first special case is the grouped section rule. If a section name has a “$” in it, then the section is called a grouped section. The linker strips the “$” and what follows to create the base name of the grouped section. The linker concatenates all of the sections in the grouped section by the alphabetical order of the characters following the “$”. For example, if an object file has sections `.idata$f`, `.idata$a`, `.idata$e`, the sections would be concatenated into a section called `.idata` containing the contents of `.idata$a` followed by the contents of `.idata$e` and `.idata$f`. This is most commonly encountered during the handling of debugging information and during the construction of the DLL imports table.

The next special case involves the placement of sections at the beginning or end of the executable. Sections with the code attribute (see Section 2.3.1) are placed first in the executable, followed by sections with the uninitialized data attribute (Table 2.5). If it exists, the `.debug` section is placed last, otherwise the `.reloc` section is placed last. This is done because the debug information is never loaded by the loader and the base relocation information (`.reloc`) section is loaded only if the executable cannot be placed in memory at its desired location.

The final special case involves object file sections that are removed by the linker and not included in the executable. These sections usually have the LINK_INFO and LINK_REMOVE attributes (Section 2.3.1). The `.drectve` section is a removable section used by the Microsoft C/C++ compiler to pass options to the Microsoft linker [11]. Figure 3.2 illustrates the concatenation process with the complete set of rules.

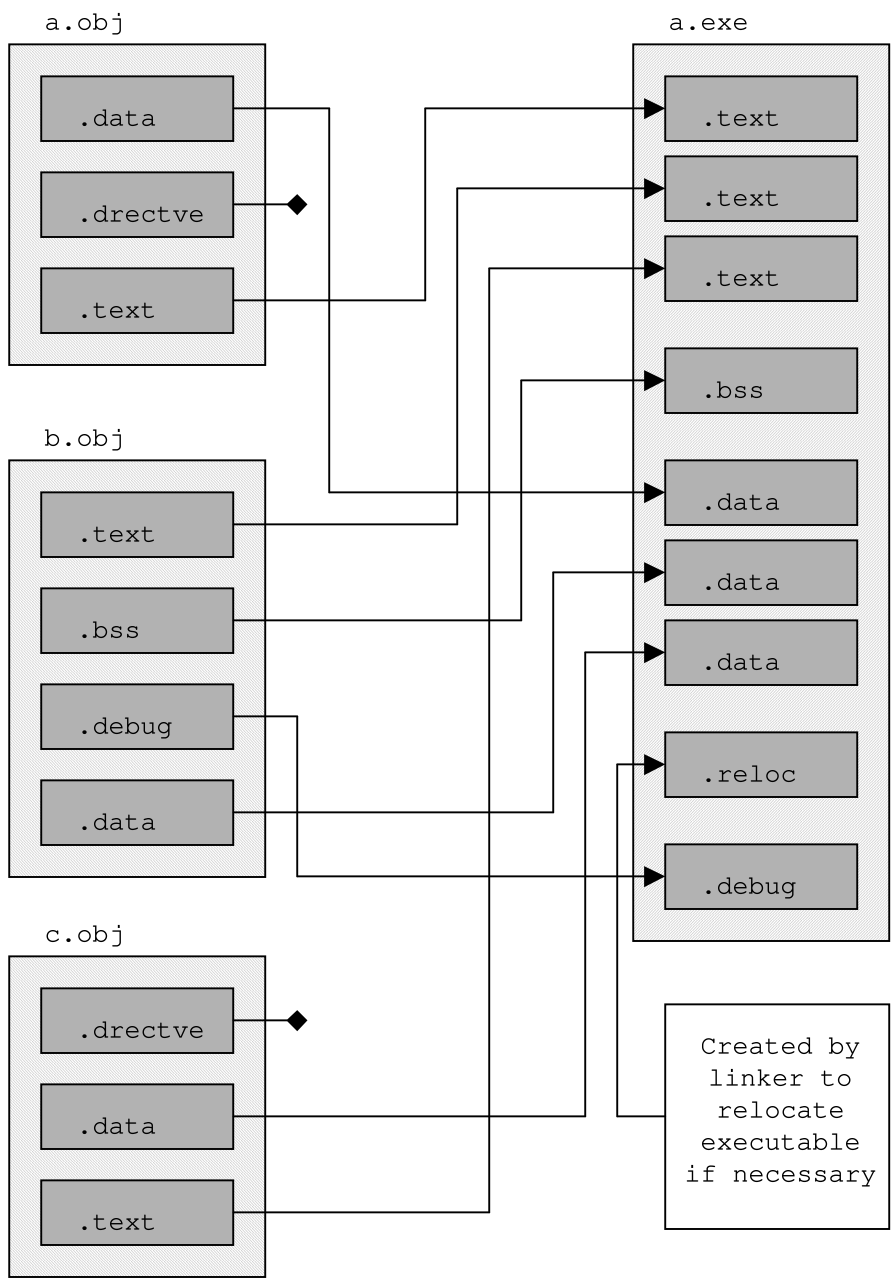

Figure 3.2 Complete Concatenation Example

3.3. Resolution

Resolution is the process of determining where an external reference is defined. When an external reference is encountered, the linker first searches the object files given on the command line and then the libraries (see Section 2.7) specified on the command line. Finally, the linker will search the libraries in the directories specified by the LIB environment variable and if the external reference cannot be located, the linker exits with an error message. An unusual rule is applied to the linking of library code. Once the definition of an external reference is located in a library, that library is given preferred status and is searched before other libraries for future references. The library loses its status once a reference is not found in the library. In this case, the next library on the list is searched [12].

Symbols that are included from a library may be referenced in two ways. First, there can be a direct reference to a symbol from one of the object files in the library. For example, if a program calls *printf()*, that use is called a direct reference to the object file containing *printf()* (in the library `libc.lib`). Second, a reference to a symbol can be an indirect reference. An indirect reference occurs when an object file included via the first method contains references to symbols in another object file. For example, `printf()` in `printf.obj` calls *ftbuf()* which is defined in `ftbuf.obj`. The file, `ftbuf.obj`, is indirectly referenced by any program that calls *printf()*. If a program contains a reference to *printf()*, the object files `printf.obj` and `ftbuf.obj` are both linked into the executable.

3.4. Linker Fixups

Compilers are usually concerned only with generating a single machine code object file from each single source code file. When the source file contains external references, the compiler does not typically know how to generate code for that function or variable reference. The solution to this problem is to leave extra information in the object file that describes where the reference is and how to use it. The term for this extra information is a "fixup" and is included in the object file as a combination of the relocation table (Section 2.3.2) and symbol table (Section 2.4). Figure 3.3 illustrates a code segment that will be used to clarify the relocation process.

```
#include <stdio.h>
void main () {
    printf("Hello\n");
}
```

Figure 3.3 Source Code for Linking Example

Relocation is the process of using the external references determined by the resolution process to "fix-up" the executable being created by the linker. To perform relocations, the linker steps through each object file and relocates every section of each object file until it has relocated every object file. The linker locates the position of the relocation by using information contained in the relocation table, the symbol table, and the section header of the section being relocated. Although many types of relocations exist, they fit into only two basic categories: direct relocation and relative relocation.

3.4.1. Direct relocation

Direct relocation is the simplest and most straightforward relocation type. Direct relocation typically occurs because of a pointer reference. Simply stated, in a direct relocation, a pointer (an absolute memory address) to an external object is placed in a location specified by the fixup record. The process is illustrated graphically in Figure 3.4 and as a code segment in Figure 3.5. The algorithm shown in Figure 3.5 is for a reference in the same object file. The algorithm for a reference to an object in another object file is similar.

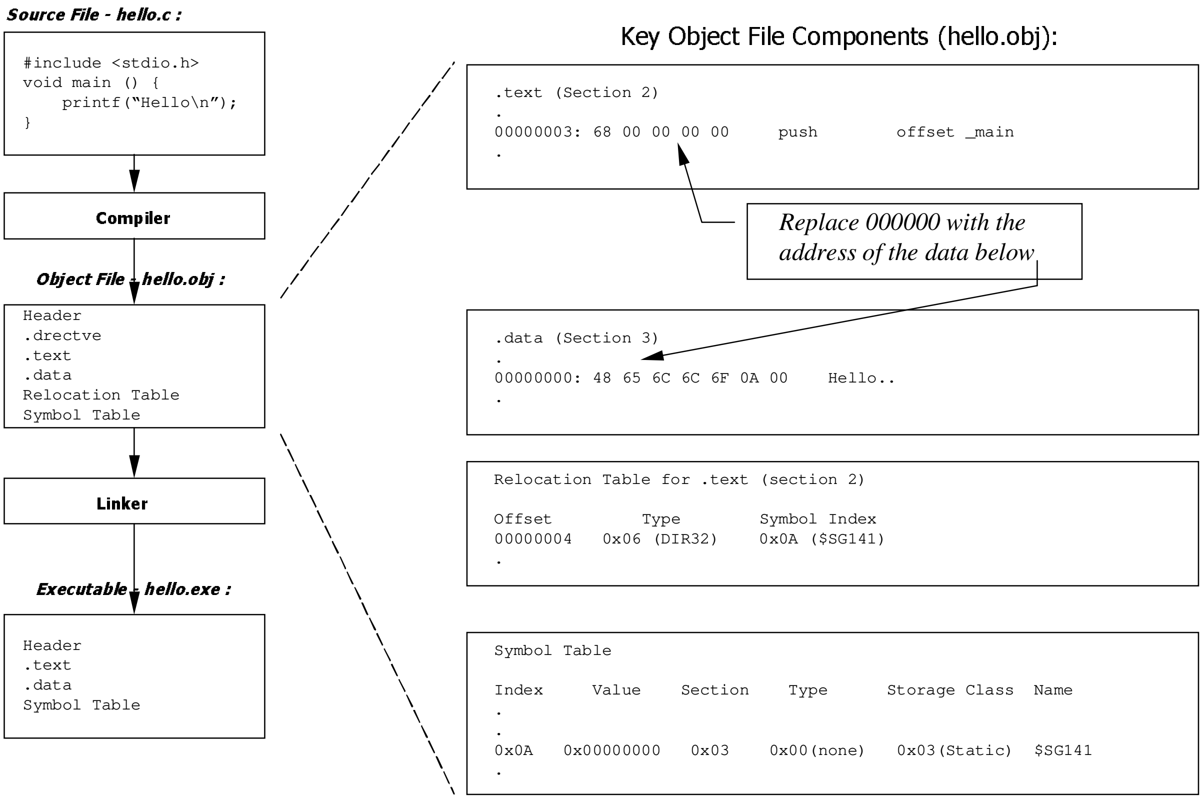

Figure 3.4 Illustration of Direct Relocation Process

```
contents of location to be relocated = address of the object + current contents of that location

location_ptr = section_table[num_of_section_being_reloced].raw + reloc_table[cur_rel_table_entry].offset;
object_addr = section_table[symbol_table[reloc_table[cur_rel_table_entry].symbol_idx]].section_num].raw
object_addr += symbol_table[reloc_table[cur_rel_table_entry].symbol_idx].Value
object_addr += *location_ptr;
*location_ptr = object_addr;
```

Figure 3.5 Algorithm for Direct Relocation

3.4.2. Relative relocation

Relative relocation is the most common relocation type. Function calls are the most common reason the compiler would create a relative relocation. In a relative relocation, the linker places the *relative offset* to an external object in the location specified by the fixup record, rather than a direct pointer. Figure 3.6 andFigure 3.7 illustrate the process graphically and algorithmically. Again, the algorithm shown in Figure 3.7 is for a reference to the same file. For an external reference, the linker would simply obtain a pointer to the function and use that as the object address.

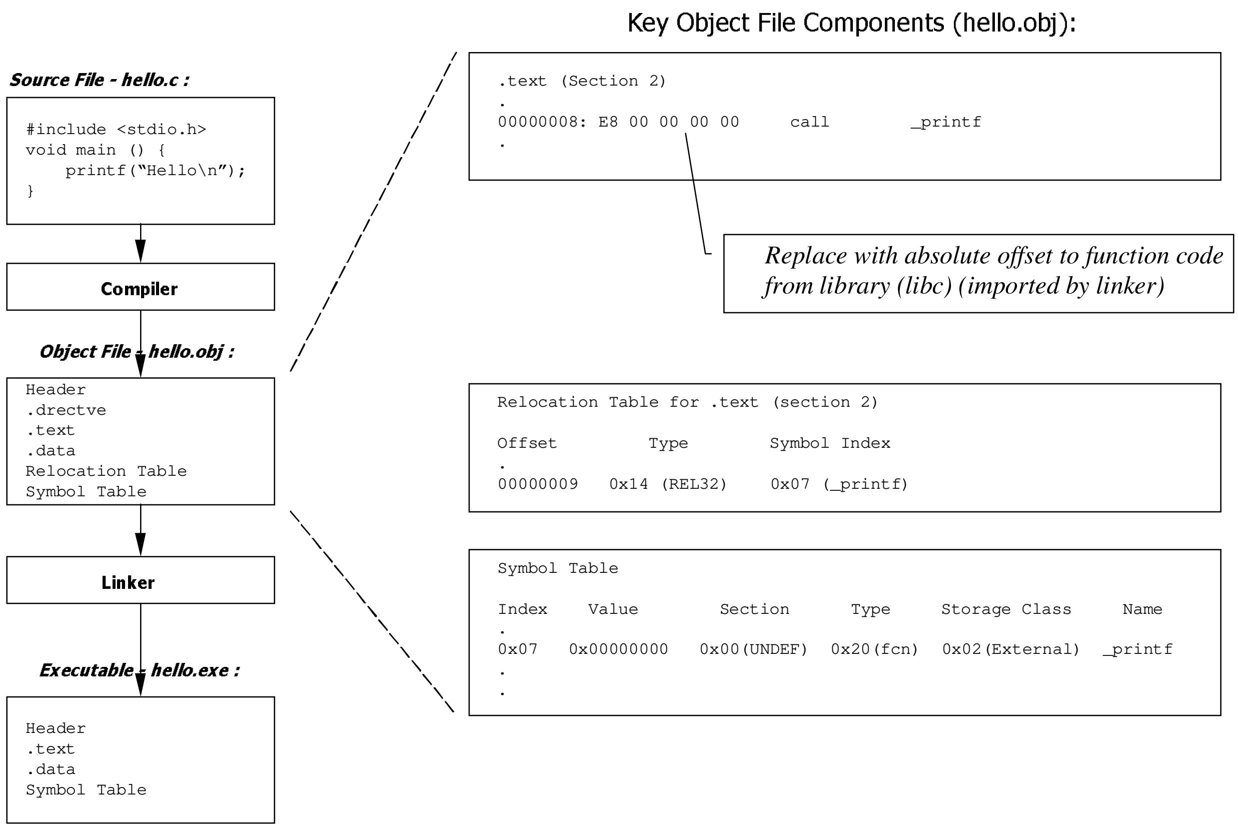

Figure 3.6 Illustration of Relative Relocation Process

```
    contents of location to be relocated = location to be relocated - address of the object - 4 bytes

location_ptr = section_table[num_of_section_being_reloced].raw + reloc_table[cur_rel_table_entry].offset;
 object_addr = section_table[symbol_table[reloc_table[cur_rel_table_entry].symbol_idx]].section_num].raw
             object_addr += symbol_table[reloc_table[cur_rel_table_entry].symbol_idx].Value
                            *location_ptr = location_ptr - object_addr - 4;
```

Figure 3.7 Algorithm for Relative Relocation

4. DESIGN GOALS AND IMPLEMENTATION

4.1. Overview

The design goals and implementation of the run-time linker is described in this chapter. The chapter begins with a section on goals and specifications (Section 4.2) and then describes the approach taken to solve the problem (Section 4.3).

4.2. Design Goals

The run-time linker was conceived to be much like a traditional linker in appearance, but extended to allow for Java's dynamic and object-oriented nature. Specifically, the linker should have an interface very similar to the standard linker, allow user-specified entry points, and allow programs to execute normally. In addition, the run-time linker should be as fast and memory efficient as possible. The linker was conceived to quickly link object files supplied within a Java runtime environment. To support this, the linker was designed to support entry point functions other than *main()* and support dynamic loading and unloading of object files. The run-time linking environment is general enough to support operations not specified in the original design. The linker can be used for quick linking during code development. The dynamic loading and unloading features make the run-time linking environment useful for run-time optimization environments or limited memory applications in the embedded processor market.

To accomplish these goals, the linker implements an innovative approach to linking. Given object files are linked in the standard way, but externally referenced symbols from libraries are dynamically loaded (and unloaded) through the Microsoft DLL interface (Section

2.8). This interface provides a standard way to perform loading and unloading using services provided by the operating system.

4.3. Implementation

The run-time linker combines the operations of a standard linker and the executable produced by such a linker. To perform these operations, the run-time linker accepts command line arguments for linking object files and executing the program contained by the object files. After processing the command line arguments, the run-time linker loads the object files given. Next, the linker loads the DLLs needed by the program to be linked. Then, the linker performs fixups on the object files and prepares them for execution. Finally, the entry point function is called and the program is executed.

4.3.1. Loading

After the command line arguments are processed and the data tables are initialized, the run-time linker loads the object files needed by the program to be executed. Each object file is loaded into a data structure that simplifies the linking process and facilitates dynamic loading and unloading of object files (Figure 4.1). As each object file is loaded, data structures used during the linking process are constructed. The first data structure is a table of loaded object files. This data structure maintains a list of loaded object files for fast lookup. The next data structure is the table of defined symbols. The linker uses this data structure to track functions and variables that have been defined. The table of defined symbols stores the symbol name and object file name. The table of object pointers is a table that relates a symbol name to the symbol's location in memory. As each symbol is discovered (defined), it is added to the table of defined symbols and the object pointer table. Finally, the table of undefined symbols contains a list of symbols that have not yet been found. These symbols are typically external references from object files. If a symbol is already a member of the table of defined symbols, it is not added to the table of undefined symbols.

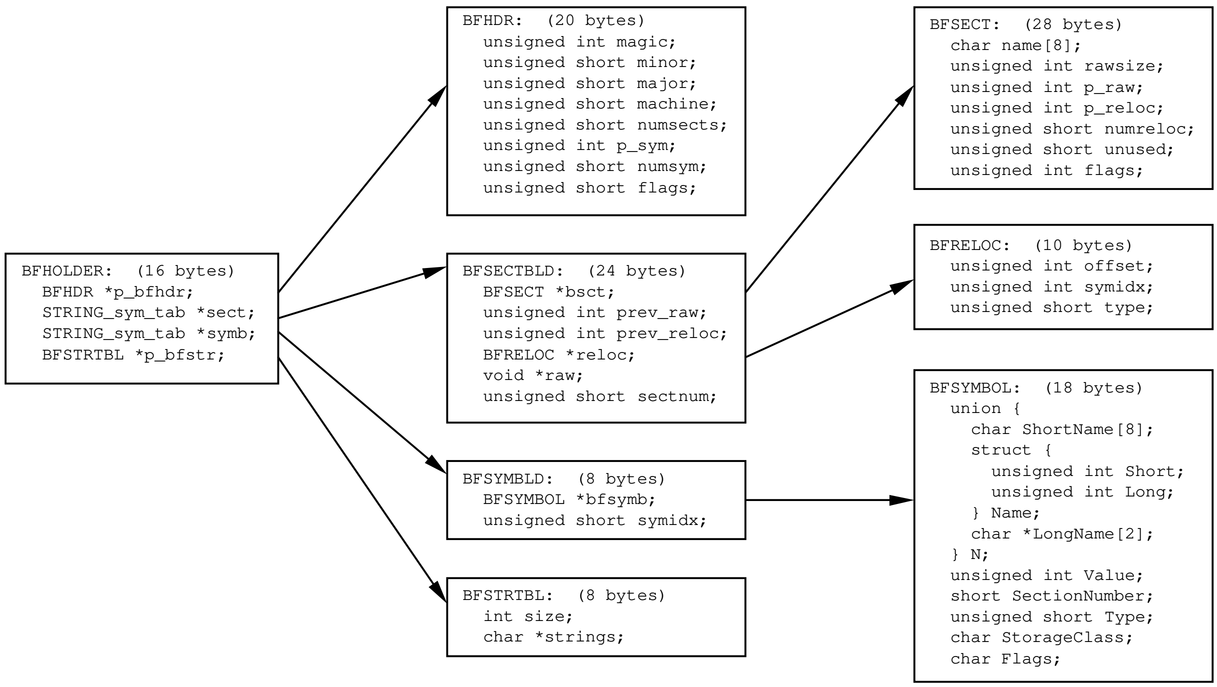

Figure 4.1 Data Structures to Hold BF Format Object Files

4.3.2. DLL Loading

Once the object files have been loaded and processed, the DLLs specified in the environment variable DLL_LIST are loaded into memory. These DLLs provide the symbols referenced by the object files. A handle for each DLL is stored in a table for later use.

4.3.3. Resolution

Resolution is the process of finding where each undefined symbol is defined. Each entry in the table of DLL handles (`p_dll_handle_table`) is queried for each symbol in the undefined symbol table. When the symbol is found it is removed from the unknown table and added to the table of known symbols and its pointer is added to the table of symbol pointers. If the undefined symbol is not found in the DLLs queried, the run-time linker exits with an error suggesting that the user add the needed DLL to the DLL_LIST environment variable.

4.3.4. Allocate memory and perform fixups

After resolving all external references, the run-time linker allocates memory for uninitialized data sections and global variables. These memory symbols are added to the table of known symbols (`p_known_sym_table`) and the table of symbol pointers (`p_fcn_ptr_table`). Next, each object file is scanned for fixups (relocations) in its data sections. Fixups are performed where necessary. Finally, code section fixups are performed by fixing up the section containing the entry point function and recursively performing fixups on each external reference.

4.3.5. Execute

The process of fixing the code sections returns a pointer to the entry point function. The entry point function is called with arguments given on the command line and the given program will execute normally.

5. USING THE RUN-TIME LINKER

5.1. Overview

This chapter provides an overview of how to use the run-time linker to link and execute programs. First, this chapter describes the process of generating the BF format object files (Section 5.2), viewing the contents of the object files (Section 5.3), and then describes how to execute the benchmark *130.li* in the run-time linking environment in Section 5.4. Finally, Section 5.5 compares the performance of the run-time linker with the Microsoft linker.

5.2. Creating BF Format Object Files

Before creating BF files, the user must first generate Microsoft COFF object files using either the Microsoft C/C++ compiler (`cl`) or other means. The `pe_to_bf` program (Appendix A) is used to convert Microsoft COFF files to the BF file format. In addition to creating BF files, `pe_to_bf` can dump the COFF file as it is read and dump the BF format file that is created for debugging. Figure 5.1 shows a simple C program (`hello.c`) that will be used in the following examples. Figure 5.2 illustrates the use of the Microsoft C compiler to create the COFF format object file, `hello.obj`, and the use of `pe_to_bf` to create the BF format object file.

```
#include <stdio.h>
void main() {
   printf("Hello World!!\n");
}
```

Figure 5.1 Example C program

```
C:\test> cl /c hello.c
Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 11.00.7022
for 80x86
Copyright (C) Microsoft Corp 1984-1997. All rights reserved.

hello.c

C:\test> pe_to_bf hello.obj
Just converted HELLO.OBJ to hello.bf
```

Figure 5.2 Use of *pe_to_bf*

5.3. Debugging BF Format Object Files

Information about BF format files can be obtained using `bf_dump`. The program `bf_dump` provides a listing of the file header, section table, raw section data, section relocation table, symbol table, and string table of a given file. Figure 5.3 illustrates the use of `bf_dump` to view the contents of a BF format file.

5.4. Executing LI in Run-time Linker

As a tutorial, this section will step through the execution of the SPEC benchmark, *130.li*, in the run-time linking environment. The benchmark *130.li* consists of 22 C source files. To use with the run-time linker, the source code must first be compiled into COFF format object files. For this example, the Microsoft C compiler, `cl`, is used for this purpose. Next, the object files are converted to BF format object files using the `pe_to_bf` program. Finally, the benchmark is ready for execution in the run-time linker, `bf_run`. The run-time linker expects a list of BF format object files as the first set of command-line arguments (standard wildcards are accepted), followed by the name of the entry-point function and any arguments expected by the program being executed. The run-time linker also accepts the /q and /l options for quiet operation and linking only, respectively. The link-only option is used for timing the link operation. These steps and their associated (abridged) output are illustrated in Figure 5.4.

```
C:\test> bf_dump hello.bf
               Dump of hello.bf:

magic:         FACEBABE
minor:         0000
major:         0000
machine:       014C
numsects:      0002
p_sym:         0081
numsym:        0006
flags:         FFFF

Section Number: 0h
Name:          .text
RawSize:       0012
PtrRaw:        004C
PtrReloc:      006D
NumReloc:      0002
Unused:        0000
Flags:         60500020

Relocation table dump of .text   :
000 Address: 00000004 SymIndex: 00000005 Type: 00000006 DIR32 $SG141
001 Address: 00000009 SymIndex: 00000003 Type: 00000014 REL32 _printf

Hex dump of raw: (.text   )
00000000:  55 8B EC 68 00 00 00 00  E8 00 00 00 00 83 C4 04 U..h............
00000010:  5D C3                                             ].

Section Number: 1h
Name:          .data
RawSize:       000F
PtrRaw:        005E
PtrReloc:      0000
NumReloc:      0000
Unused:        0000
Flags:         C0300040

Hex dump of raw: (.data   )
00000000:  48 65 6C 6C 6F 20 77 6F  72 6C 64 21 21 0A 00    Hello world!!..

Symbol Table - 6 entries
Indx Name                 Value    Section    Type    Storage
---- -------------------- -------- ---------- ------- --------
0000 .file                00000000 sect:DEBUG type:00 st:67
0001 .text                00000000 sect:1     type:00 st:3
0002 _main                00000000 sect:1     type:20 st:2
0003 _printf              00000000 sect:UNDEF type:20 st:2
0004 .data                00000000 sect:2     type:00 st:3
0005 $SG141               00000000 sect:2     type:00 st:3
```

Figure 5.3 Using `bf_dump`

```
C:\test> cl /nologo /c *.c
OS.C
XLBFUN.C
. . .
XLSYS.C
Generating Code...

C:\test> pe_to_bf *.obj
Just converted OS.obj to os.bf
Just converted XLBFUN.obj to xlbfun.bf
. . .
Just converted XLSYS.obj to xlsys.bf

C:\test> bf_run *.bf main  < li-input.lsp
DLL_LIST is not set!
XLISP version 1.6, Copyright (c) 1985, by David Betz
> > > > > > > > > 1> CADAR
> > > 1> 2> 2> 2> THREAT
> > > 1> 2> 2> CONFLICT
> > > > 1> 2> 2> 2> 2> 2> 2> 2> 2> 4> 2> 2> 2> 3> 4> 4> 2> 2>
2> 4> 3> QUEENS
. . .
((1 1) (2 3) (3 8) (4 6) (5 9) (6 2) (7 5) (8 7) (9 4))
((1 1) (2 4) (3 2) (4 8) (5 6) (6 9) (7 3) (8 5) (9 7))
DONE
>
```

Figure 5.4 Executing LI in the run-time linker

5.5. Comparing the Run-time Linker with the Microsoft Linker

To effectively benchmark the run-time linker against the Microsoft linker, two synthetic benchmarks and the SPEC benchmarks *130.li*, *099.go*, *008.espresso*, and *124.m88ksim* are used. In all cases, the run-time linker outperformed the Microsoft linker. It is important to note that the run-time linker also performs load time operations such as memory allocation for uninitialized data and global variables in addition to linking.

For each sample point, 110 samples are taken. The first two samples are discarded to remove file cache effects. Next, the samples are sorted and the top five and bottom five execution times are discarded. Finally, the remaining 98 points are averaged and the result is reported.

5.5.1. Synthetic benchmark 1: number of files

The first synthetic benchmark tests each linker's ability to handle a large number of input files. The benchmark consists of a main program file that calls the subroutine in each subroutine file. For testing 10 files, the linker must link a "main" object file that calls 10 different subroutines with 10 separate subroutine object files containing one subroutine each for a total of 11 files. Figure 5.5 compares the run-time linker to the Microsoft linker for the "Number of Files" benchmark.

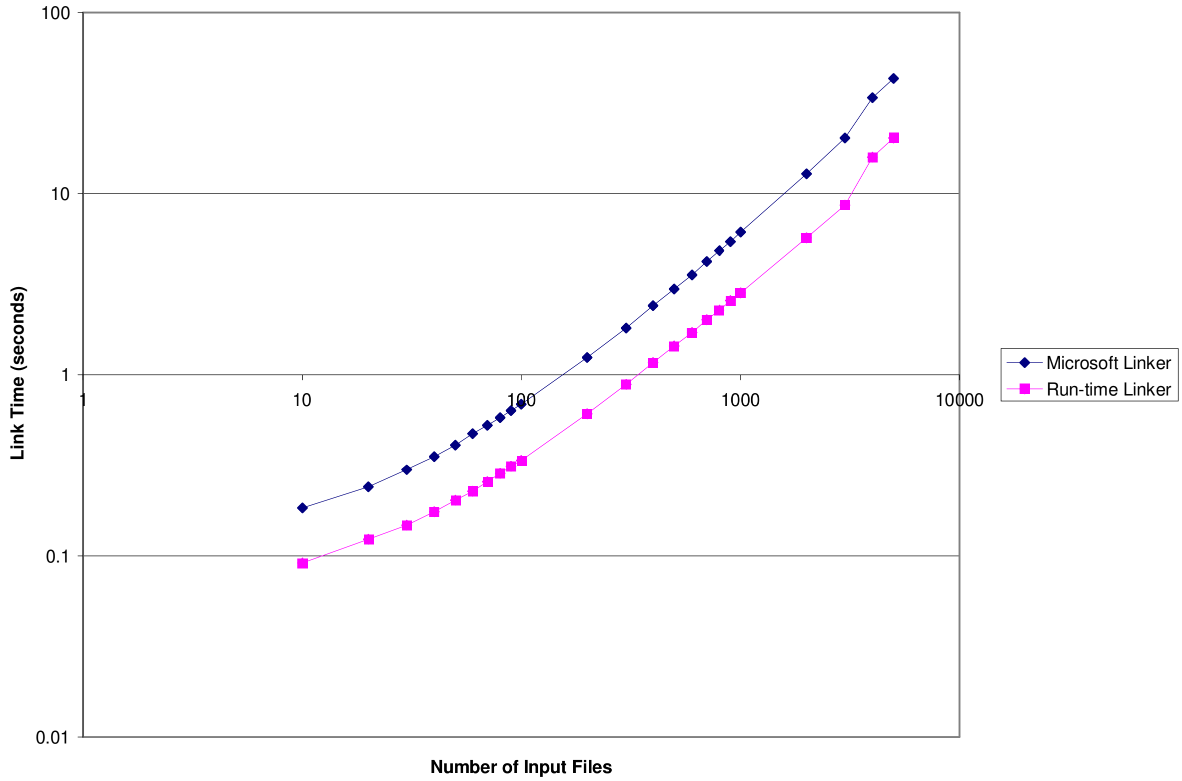

Figure 5.5 Link time vs. Number of Input Files

Figure 5.5 clearly illustrates that the run-time linker outperforms the Microsoft linker for the test cases illustrated. A simple curve fit will find that the Microsoft linker and the run-time linker both have O(N log(N)) behavior. This is likely due to the application of similar hashing schemes.

The benchmarks were also run on a ramdisk to determine if the hard drive affected the results (unlike the Microsoft linker, the run-time linker does not write an executable to disk)[2]. The ramdisk results were nearly identical to the hard drive results and were actually worse in some cases. For these reasons, these results are omitted.

[2] The ramdisk used was an evaluation copy of RamDiskNT v1.1-0 from EEC Systems, www.eecsys.com.

5.5.2. Synthetic benchmark 2: number of functions

The next synthetic benchmark is the "number of functions" benchmark. For this benchmark, a "main" object file calls a given number of functions in a subroutine file. To test 10 functions, for example, the linker must link a main object file that calls 10 subroutines with a subroutine file that contains 10 subroutine functions. The results of the "number of functions" benchmark is shown in Figure 5.6.

For the number of functions benchmark, the run-time linker outperforms the Microsoft linker again. For this benchmark, the run-time linker executes in O(N) time, while the Microsoft linker executes in near constant time until 3000 functions are given when the curve becomes $O(n^2)$. As observed in Figure 5.6, the linking overhead is lower when there are many functions in each file.

5.5.3. SPEC Benchmarks

The SPEC benchmarks *130.li*, *099.go*, *008.espresso*, and *124.m88ksim* were selected to test the performance of the run-time linker for "real" applications. These benchmarks range from 17 to 97 object files. The link times of these benchmarks are shown in Figure 5.7.

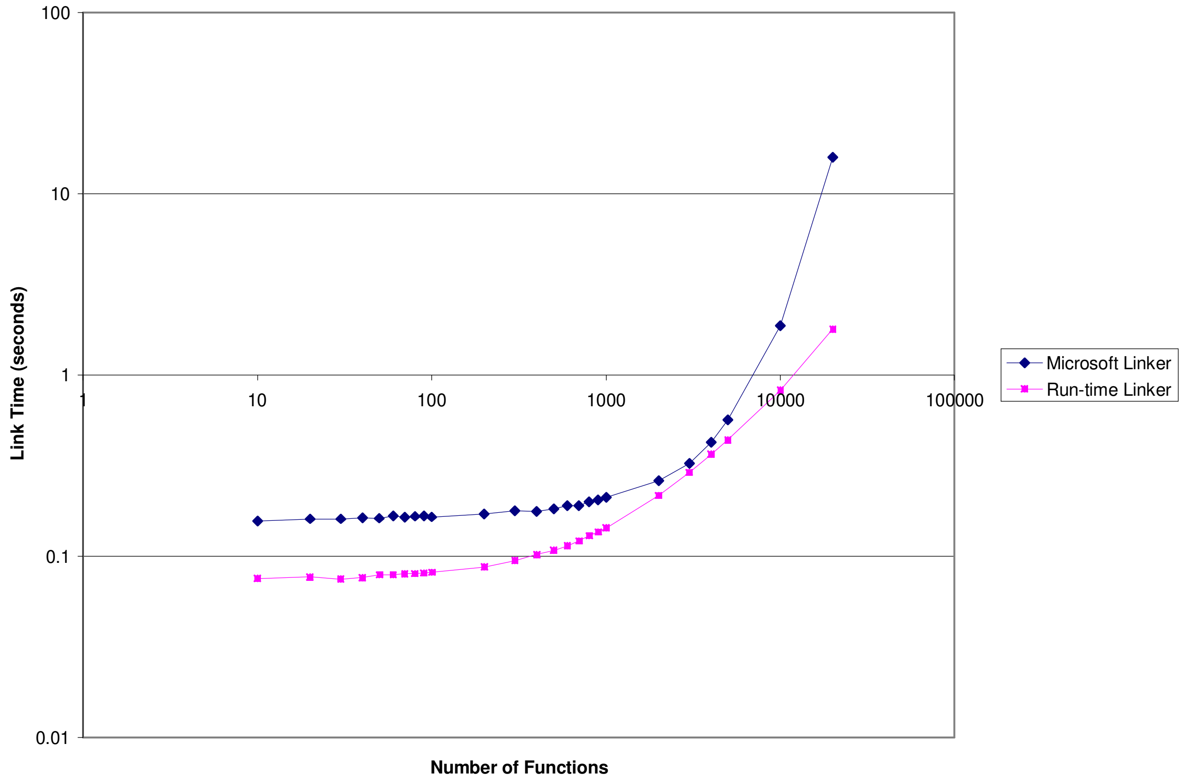

Figure 5.6 Link Time vs. Number of Functions

These results show that the run-time linker is still faster than the Microsoft linker for "real" applications. These results also show that the link time required before execution is small enough to be considered negligible when compared to the execution time of most applications.

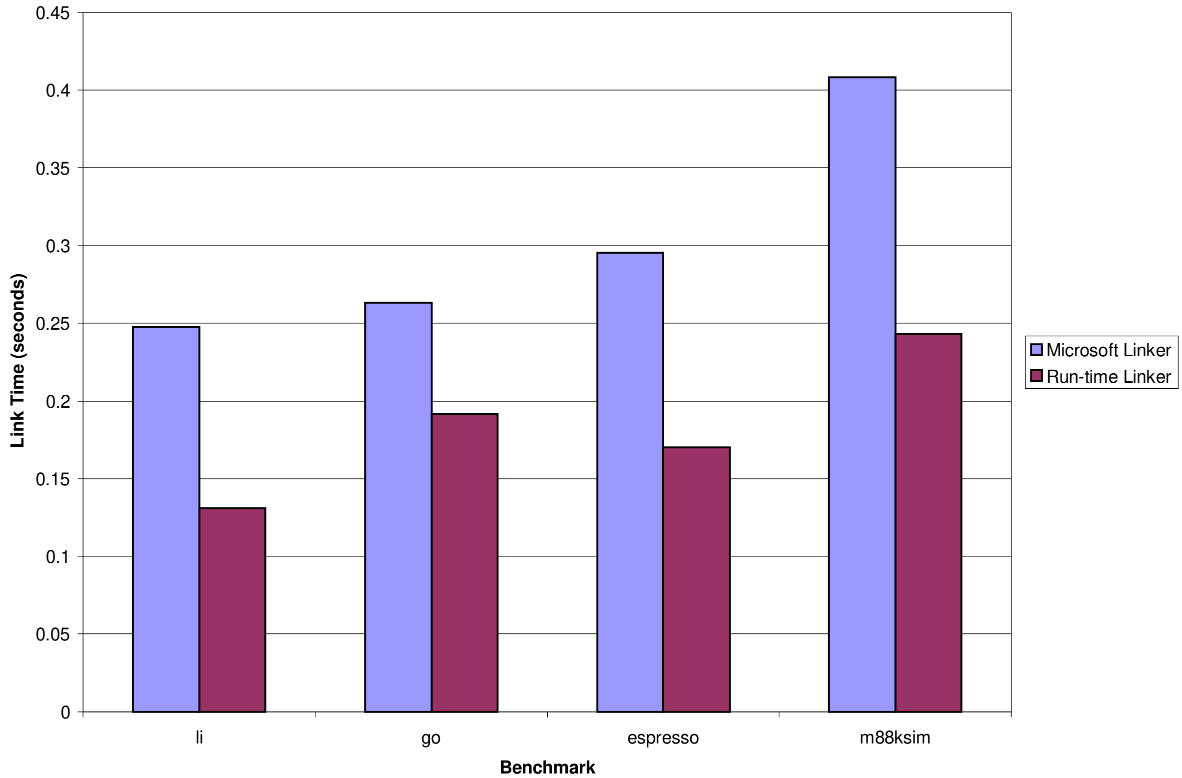

Figure 5.7 Link Times for Selected SPEC Benchmarks

6. SUMMARY

This thesis proposes a powerful method for quickly linking and executing object files in a dynamic code environment. The current implementation of the run-time linker and the COFF conversion utilities work for the common cases and provide a basis for future work in run-time linking.

The first extension would be to complete the run-time linker's handling of more complex linking. This would allow the run-time linker to handle as many and varied conditions as the Microsoft linker. Support for all corner cases and additional languages would allow more experimentation in this developing field.

The second extension would be to provide incremental run-time linking. In this case, the object files would be patched to call into the linker at each external function call. Program execution would begin immediately at the entry point function and linking would occur as needed during the execution of the program. Perhaps a modified run-time linking system could be integrated with a debugger to allow the user to remove buggy object modules and replace them with debugged ones without exiting the debugging environment.

Another proposed extension would be to allow the run-time linking environment to support dynamic loading and unloading. This would allow more research into the use of dynamic compilation on memory constrained systems.

APPENDIX A: OBJECT FILE CONVERSION

The program pe_to_bf was written to perform the conversion from the Microsoft COFF format to the BF object file format. The execution syntax of the program is straightforward and simple. The pe_to_bf program supports wildcard expansion and has switches useful in debugging.

```
pe_to_bf - COFF .OBJ to .bf converter - Build 2/25/98
(C) 1997 Rich Kutter

Syntax:  pe_to_bf [/A | [/B | /D | /H]] pefiles.obj
        /A      Dump All
        /B      Dump resulting BF file
        /D      Dump PE format Object file
        /H      Hex Dump PE Object file
```

APPENDIX B: ERRATA FOR MICROSOFT PE AND COFF SPECIFICATION 4.1

Section 5.2.1:

The Value for Constant IMAGE_REL_I386_DIR32 is 6, not 3 and it is a Direct 32-bit reference to the symbol's virtual address, *plus* the offset given by the Relocation Table.

Section 5.6:

The size field of the COFF String Table includes the 4 bytes of the size field itself.

Section 7.3:

The name of the first linker member is "/", not "\".

Section 7.4:

The name of the second linker member is "/", not "\".

Section 7.5:

The name of the longnames member is "//", not "\\".

The longnames header is not required to appear in the archive.

REFERENCES

[1] T. Lindholm, F. Yellin, B. Joy, and K. Walrath, *The Java Virtual Machine Specifications*. Reading, MA: Addison-Wesley, 1996.

[2] *Portable Executable and Common Object File Format Specification 4.1*. Microsoft Corporation, 1994.

[3] Y. Andoh, "j2c," October, 1996, http://www.webcity.co.jp/info/andoh/java/j2c.html.

[4] T. A. Proebsting et al., "Toba: Java for applications a way ahead of time (WAT) compiler," in *Proceedings of the Third USENIX Conference on Object-Oriented Technologies and Systems*, 1997, pp 30-43.

[5] C. A. Hsieh, J. C. Gyllenhaal, and W. W. Hwu, "Java bytecode to native code translation: the caffeine prototype and preliminary results," in *Proceedings of the 29th Annual ACM/IEEE International Symposium of Microarchitecture*, 1996, pp 90-100.

[6] M. Franz, "Dynamic linking of software components," *IEEE Computer*, vol. 30, pp. 74-81, March 1997.

[7] A. Acharya and J. Saltz, "Compiler-directed dynamic linking for mobile programs," in *Mobile Object Systems*, J. Vitek and C. Tschudin (Eds.), Heidelberg: Springer Verlag Lecture Notes in Computer Science, 1996.

[8] R. A. Gingell, M. Lee, X. T. Dang, and M. S. Weeks, "Shared libraries in Sun OS," *Usenix Conference Proceedings*, pp. 131-145, Summer 1987.

[9] "winnt.h" Source code file distributed with Microsoft Visual C++, 1997.

[10] Microsoft Corporation, "Microsoft Developers Network Library Online," 1997, http://premium.microsoft.com/msdn/library/?FinishURL=/msdn/library/.

[11] M. Pietrek, "Under the hood," *Microsoft Systems Journal*, vol. 108, July 1997, pp. 77-83.

[12] Microsoft Knowledge Base Article Number Q31998, Microsoft Corporation, 1995.