**Portable Systems Group**

# NT OS/2 Linker/Librarian/Image Format Specification

**Author:** *Michael J. O'Leary*

*Revision 1.3, May 31, 1990*

## 1. Overview

This specification describes the Linker and Librarian for the NT OS/2 system. The Common Object File Format (COFF) standard with extensions needed to support Dynamic Linked Libraries (DLL's) and new languages such as C++ will be used both as the Object Module Format (OMF) produced by the compilers/assemblers and the executable image format used by the operating system to load a program.

### 1.1 Design Goals

- o Fastest possible image activation.
- o Minimize and localize pages that can't be shared and require fixups.
- o Able to base a DLL or image at a prefered memory location.
- o Linker is the only program that modifies or constructs images.
  - o Resource compiler will produce object fed to linker.
- o Need to easily support extensions to image format.
  - o Linker will support multiple sections in objects.

### 1.2 Constraints

- o Must be able to distinguish Cruiser Images vs NT images.
  - o Header must have common flags.
- o DLL support compatible with Cruiser.
  - o Support transfer of control (calls) and data references.
  - o All init routines called before program entry.
- o Must be compatible with Intel i860 assembler.
  - o Understand basic coff.
  - o Identify Intel extensions.

# 2. Coff

## 2.1 What is Coff?

Coff (Common Object File Format) is the formal definition for the structure of machine code files in the UNIX System V environment. All machine code files, whether fully linked executables, compiled applications, or system libraries, are COFF structured files. This will also become the formal definition for NT OS/2.

The COFF definition describes a complex data structure that represents object files, executable files, and archive (library) files. The Coff data structure defines fields for machine code, relocation information, symbolic information, and more. The contents of these fields are accessed by an organized system of pointers. Assemblers, compilers, linkers, and archivers manipulate the contents of the COFF data structure to achieve their particular objective.

## 2.2 Why Coff?

Coff was chosen over the Crusier Linear Executable Format because of the following reasons.

- o Crusier images are not mappable.
    - o No mappable image header.
    - o Text and data pages are not laid out in the file such that they can be direclty mapped and paged into memory. Must grovel over a mapping table to determine page table contents.
    - o Preloaded pages prohibit mapping.
    - o Certain fields are not on their natural alignments.
    - o Iterated data pages prohibit mapping.
- o Crusier format contains 386 specifics.
- o Wasted space for fields that will never be used.
    - o Verify Record Table.
    - o Resident Name Table.
    - o Checksums.

- Fixups are by page/offset instead of by virtual address.
- Resource Compiler modifies executable image.
- Current i860 tools support COFF. We don't want to have to do another assembler.

## 2.3 Coff Structure

### 2.3.1 Coff File Layout

For NT OS/2, the following diagram shows the structure of a basic coff file. All headers must be at the beginning of the file. All other parts of the file can be in any order. An executable file will always be in the order show in this diagram.

```
        Ö-----------------------------───────┐
virtual ° FILE HEADER                        °  relative
pointers°      TargetMachine                 °  sizes
        °      NumberOfSections--------------û-Ì
        °      TimeDateStamp                 ° °
Ö-------À------PointerToSymbolTable          ° °
°       °      NumberOfSymbols---------------û-é-------Ì
°       °      SizeOfOptionalHeader----------ûÌ°       °
°       °      Characteristics               °°°       °
°       û-----------------------------──────-À°°       °
°       ° OPTIONAL HEADER                    °°°       °
│       │      TargetVersionStamp            │││       │
│       │      LinkerVersionStamp            │││       │
│       │      SizeOfCode                    │││       │
│       │      SizeOfInitializedData         │││       │
│       │      SizeOfUninitializedData       │││       │
│       │      AddressOfEntryPoint           │││       │
│       │      BaseOfCode                    │││       │
│       │      BaseOfData                    │││       │
│       │      ImageBase                     │││       │
│       │      TargetOperatingSystem         │││       │
│       │      TargetSubsystem               │││       │
│       │      ImageVersionStamp             │││       │
│       │      SizeOfImage                   │││       │
│       │      SizeOfHeaders                 │││       │
│       │      SizeOfHeap                    │││       │
│       │      SizeOfHeapCommit              │││       │
│       │      SizeOfStack                   │││       │
│       │      SizeOfStackCommit             │││       │
│       │      ZeroBits                      │││       │
│       │      CheckSum                      │││       │
│┌──────┤------PointerToBaseRelocations      │││       │
││      │      NumberOfBaseRelocations-------├┼┼──┐    │
││      │      AddressOfProcessInitRoutine   │││  │    │
││      │      AddressOfThreadInitRoutine    │││  │    │
││      │      AddressOfDllTable             │││  │    │
││      │      SectionNumberByType[6]        ├┘│  │    │
││      │      AdditionalMachineValues[8]    │ │  │    │
°│      û-----------------------------──────-À °  │    °
°│      ° SECTION HEADER                     ° °  │    °
°│      °      Name (e.g.,.text)             ° °  │    °
```

```
°|      °      PhysicalAddress               ° °  |    °
°|      °      VirtualAddress                ° °  |    °
°|      °      SizeOfRawData-----------------û-é-Ì|    °
°|  Ö---À------PointerToRelocations          ° ° °|    °
°|  ° Ö-À------PointerToRawData              ° ° °|    °
°|Ö-é-é-À------PointerToLineNumbers          ° ° °|    °
°|° ° ° °      NumberOfRelocationEntries-----û-é-é┼Ì   °
°|° ° ° °      NumberOfLineNumberEntries-----û-é-é┼é-Ì °
°|° ° ° °      Characteristics               ° ° °|° ° °
°|° ° ° û------------------------------──────À ° °|° ° °
°|° ° ° ° other section header               ° ° °|° ° °
°|° ° ° û------------------------------──────À ° °|° ° °
°|° ° ° ° last section header                û-ì °|° ° °
°|° ° ° û------------------------------──────À   °|° ° °
°└┼─┼─┼─┤ base relocations                   |   °|° ° °
° ° ° ° °                                    ├───┼┘° ° °
° ° ° ° û------------------------------──────À   ° ° ° °
° ° ° Û-À raw data (.text)                   °   ° ° ° °
° ° °   °                                    û---ì ° ° °
° ° °   û------------------------------──────À     ° ° °
° ° °   ° other sections raw data            °     ° ° °
° ° °   û------------------------------──────À     ° ° °
° ° Û---À first relocation entry             °     ° ° °
° °     °      virtual address               °     ° ° °
° °     °      symbol table index            °     ° ° °
° °     °      relocation type               °     ° ° °
° °     û------------------------------──────À     ° ° °
° °     ° last relocation entry             û-----ì ° °
° °     û------------------------------──────À       ° °
° °     ° other sections relocations         °       ° °
° °     û------------------------------──────À       ° °
° Û-----À first line number entry            °       ° °
°       °      symbol table index            °       ° °
°       °      line number                   °       ° °
°       û------------------------------──────À       ° °
°       ° last line number entry            û-------ì °
°       û------------------------------──────À         °
°       ° other sections line numbers        °         °
°       û------------------------------──────À         °
Û-------À symbol table                       °         °
        °      name or string pointer        °         °
        °      virtual address               °         °
        °      section number                °         °
        °      type                          °         °
        °      class                         °         °
        °      number aux entries            °         °
        °                                   û---------ì
        û------------------------------──────À
        ° [size] string table                ° SymPtr+NumSyms*SizeSym
        Û------------------------------──────ì
```

### 2.3.2 Coff File Header

The file header size and format is that of standard COFF.

```
typedef struct _FILE_HEADER {
        USHORT TargetMachine;
        USHORT NumberOfSections;
        ULONG  TimeDateStamp;
        ULONG  PointerToSymbolTable;
        ULONG  NumberOfSymbols;
        USHORT SizeOfOptionalHeader;
        USHORT Characteristics;
} FILE_HEADER, *PFILE_HEADER;
```

<u>FILE_HEADER Structure:</u>

*TargetMachine* —Indicates the target machine the object/image file is executable.

**<u>TargetEnvironment Flags</u>**:

**COFF_FILE_TARGET_UNKNOWN** —Indicates unknown target machine.

**COFF_FILE_TARGET_860** —Indicates the object/image is binary compatable with the Intel i860 instruction set.

**COFF_FILE_TARGET_386** —Indicates object/image is binary compatable with the Intel 386 instruction set.

**COFF_FILE_TARGET_MIPS** —Indicates object/image is binary compatable with the Mips instruction set.

*NumberOfSections* —Indicates the number of section headers contained in the file. The number of the first section is one.

*TimeDateStamp* —Indicates the time and date when the file was created. Number of elapsed seconds since 00:00:00 GMT, January 1, 1970.

*PointerToSymbolTable* —A file pointer (offset from the beginning of the file) to the start of the symbol table. The symbol table is sector aligned on disk.

*NumberOfSymbols* —Indicates the number of symbol table entries. Each entry is 18 bytes in length.

*SizeOfOptionalHeader* —Indicates the size of the optional header.

*Characteristics* —Indicates the characteristics of the object file.

**Characteristics Flags**:

**COFF_FILE_RELOCS_STRIPPED** —Relocation information stripped from file.

**COFF_FILE_EXECUTABLE_IMAGE** —No unresolved external references.

**COFF_FILE_LINE_NUMS_STRIPPED** —Line numbers stripped from file.

**COFF_FILE_LOCAL_SYMS_STRIPPED** —Local symbols stripped from file.

**COFF_FILE_MINIMAL_OBJECT** —Reserved.

**COFF_FILE_UPDATE_OBJECT** —Reserved.

**COFF_FILE_BYTES_REVERSED** —Bytes of machine word are reversed.

**COFF_FILE_MACHINE_16BITS** —16 bit word machine.

**COFF_FILE_MACHINE_32BITS** —32 bit word machine.

**COFF_FILE_PATCH** —Reserved.

**COFF_FILE_NT_EXTENSIONS** —If set, specifies the file contains new section headers and padded symbol table.

**COFF_FILE_DLL** —Image is a Dynamic Link Library.

**COFF_FILE_BYTES_REVERSED_LO** —Bytes of machine are reversed.

**COFF_FILE_BYTES_REVERSED_HI** —Bytes of machine are reversed. You can test either of the above two bits, they are in the same bit position in each short word. This allows you to identify if the coff object/image was written for a big or little endian machine.

### 2.3.3 Coff Optional Header

There is no standard COFF optional header size and format. NT defines the optional header as:

```
typedef struct _OPTIONAL_HEADER {
        USHORT TargetVersionStamp;
        USHORT LinkerVersionStamp;
        ULONG  SizeOfCode;
        ULONG  SizeOfInitializedData;
        ULONG  SizeOfUninitializedData;
        ULONG  AddressOfEntryPoint;
        ULONG  BaseOfCode;
        ULONG  BaseOfData;
        ULONG  ImageBase;
        USHORT TargetOperatingSystem;
        USHORT TargetSubsystem;
        ULONG  ImageVersionStamp;
        ULONG  SizeOfImage;
        ULONG  SizeOfHeaders;
        ULONG  SizeOfHeap;
        ULONG  SizeOfHeapCommit;
        ULONG  SizeOfStack;
        ULONG  SizeOfStackCommit;
        ULONG  ZeroBits;
        ULONG  CheckSum;
        ULONG  AddressOfBaseRelocations;
        ULONG  NumberOfBaseRelocations;
        PVOID  AddressOfProcessInitRoutines;
        PVOID  AddressOfThreadInitRoutines;
        ULONG  AddressOfDllTable;
        USHORT SectionNumberByTYpe[6];
        ULONG  AdditionalMachineValues[8];
} OPTIONAL_HEADER, *POPTIONAL_HEADER;
```

OPTIONAL_HEADER Structure:

*TargetVersionStamp* —Indicates operating system version.

*LinkerVersionStamp* —Indicates which version of the linker was used to build image.

*SizeOfCode* —Indicates the number of bytes of code.

*SizeOfInitializedData* —Indicates the number of bytes of initialized data.

*SizeOfUnInitializedData* —Indicates the number of bytes of uninitialized data.

*AddressOfEntryPoint* —Relative virtual address of starting point of image. This value added to the ImageBase is the virtual address of the entrypoint.

*BaseOfCode* —Indicates the relative virtual address (64K aligned) of the origin of the first byte of code. This value added to the ImageBase is the virtual address of the code.

*BaseOfData* —Indicates the relative virtual address (64K aligned) of the origin of the first byte of data. This value added to the ImageBase is the virtual address of the data.

*ImageBase* —Indicates the virtual address (64K aligned) of the origin of the file header.

*TargetOperatingSystem* —Indicates operating system and system version on which the image is executable.

**TargetOperatingSystem Flags**:

**COFF_OPTIONAL_TARGET_OS_UNKNOWN** —Indicates unknown target environment.

**COFF_OPTIONAL_TARGET_OS_NTOS2** —Indicates image is targeted for NT OS/2.

*TargetSubsystem* —Indicates which subsystem of the operating system the image is intended to run under.

**TargetSubsystem Flags**:

**COFF_OPTIONAL_TARGET_SUBSYSTEM_UNKNOWN** —Indicates unknown subsystem.

**COFF_OPTIONAL_TARGET_SUBSYSTEM_NATIVE** —Indicates image runs under the native operating system. Subsystems are native images.

**COFF_OPTIONAL_TARGET_SUBSYSTEM_OS2** —Indicates image is to run in the OS/2 subsystem.

**COFF_OPTIONAL_TARGET_SUBSYSTEM_POSIX** —Indicates image is to run in the Posix subsystem.

*ImageVersionStamp* —Indicates image version. To be used for backword compatibility. This stamp can be set by the user with the Version: switch.

*SizeOfImage* —Indicates the virtual size of the image.

*SizeOfHeaders* —Indicates the total size of all headers.

*SizeOfHeap* —Indicates the maximum size the heap is allowed to grow.

*SizeOfHeapCommit* —Indicates the initial heap size.

*SizeOfStack* —Indicates the maximum size the stack is allowed to grow.

*SizeOfStackCommit* —Indicates the initial stack size.

*ZeroBits* —Indicates how memory is to be allocated.

*PointerToBaseRelocations* —A file pointer to a table that is used to apply relocations to the image if the image can't be based at its desired base location. The first long word of the base table indicates the number of base table entries that follow. PointerToBaseTable will be zero if the image doesn't have a base table. The base table structure is defined later in this document.

*AddressOfProcessInitRoutines* —TBD.

*AddressOfThreadInitRoutines* —TBD.

*AddressOfDllTable* —The relative virtual address of a table that defines DLL's. This is described later in this document.

*SectionNumberByType* —Is any array of interesting section numbers.

**SectionNumberByType index values**:

**COFF_SECTION_TYPE_DEBUG** —Indicates the section with contains the debug information.

**COFF_SECTION_EXPORTS** —Indicates the section with contains the export table.

**COFF_SECTION_RESOURCE** —Indicates the section with contains the resource data.

**COFF_SECTION_SECURITY —**Indicates the section with contains security information.

**COFF_SECTION_EXCEPTION —**Indicates the section with contains the exception tables.

The optional header is used only for images. If an object file contains an optional header of the proper size, it is used in the following manner:

If *TargetSubsystem* is not **COFF_OPTIONAL_TARGET_SUBSYSTEM_UNKNOWN**, then a subsystem is being defined. It tells the linker that the following sections within this file are for a particular subsystem. With this information, the linker can guarantee that different subsystem components won't be mixed together. Each library should contain one of these records.

If *AddressOfEntryPoint* is non-zero, then an entrypoint is being defined. This allows a compiler to supply the entrypoint without using the linker command line switch.

All other fields are ignored.

### 2.3.4 Coff Section Header

All section headers must follow the file header (or optional header if there is one).

An object or image can contain any number of sections and in any order. The linker combines any sections with the same name and with the same flags. For example, if a compiler wants to keep all constants together, then the compiler could use a section name of .const in every object that contained constants. The linker will merge these sections together (provided they also had the same flag attribute such as R/O). In some coff implementations, if a section is empty (i.e., object contains no .bss), a section header still identifies the section, but would contain a zero size. For NT OS/2, this extra section header is not required.

Section names must start with a period (.). For each section, a special symbol will be defined by the linker. The period (.) will be replaced with a colon (:). This will be the next address after the section. Thus if a section is named **.text**, then the linker will create the symbol **:text**.

Grouping of sections hasn't been determined yet.

There are two styles of the section header. The first section header size and format is that of standard COFF. The second section header is an extension added to Coff. Both headers are the same size, but different format. The

**COFF_OPTIONAL_NT_EXTENSIONS** flag in the file header specifies which section header the object contains. Section headers can not be mixed within one object, they must all be of one type. The image file will always have the **COFF_OPTIONAL_NT_EXTENSIONS** flag set, and thus the image will always contain new section headers.

The standard Coff section header has the following format:

```
typedef struct _OLD_SECTION_HEADER {
        UCHAR    Name[8];
        ULONG    PysicalAddress;
        ULONG    VirtualAddress;
        ULONG    SizeOfRawData;
        ULONG    PointerToRawData;
        ULONG    PointerToRelocations;
        ULONG    PointerToLinenumbers;
        USHORT   NumberOfRelocations;
        USHORT   NumberOfLineNumbers;
        ULONG    Characteristics;
} OLD_SECTION_HEADER, *POLD_SECTION_HEADER;
```

The new section header the following format:

```
typedef struct _NEW_SECTION_HEADER {
        UCHAR    Name[8];
        ULONG    NumberOfLinenumbers;
        ULONG    VirtualAddress;
        ULONG    SizeOfRawData;
        ULONG    PointerToRawData;
        ULONG    PointerToRelocations;
        ULONG    PointerToLinenumbers;
        ULONG    NumberOfRelocations;
        ULONG    Characteristics;
} NEW_SECTION_HEADER, *PNEW_SECTION_HEADER;
```

<u>SECTION_HEADER Structure:</u>

*Name* —Eight character null padded section name.

*PysicalAddress* —Indicates the physical address of the section. This field only exits within the old section header. Its value is never used.

*VirtualAddress* —Indicates the relative virtual address of the section.

*SizeOfRawData* —Indicates the size in bytes of the sections raw data.

*PointerToRawData* —A file pointer (offset from the beginning of the file) to the raw data for this sections.

*PointerToRelocations* —A file pointer (offset from the beginning of the file) to the relocation entries for this section. The relocation entries are sector aligned on disk.

*PointerToLinenumbers* —A file pointer (offset from the beginning of the file) to the line number entries for this section. The line number entries are sector aligned on disk.

*NumberOfRelocations* —Indicates the number of relocation entries for this section.

*NumberOfLinenumbers* —Indicates the number of line number entries for this section.

*Characteristics* —This flag represent three kinds of information:

- Section Type
- Section Content
- Section Memory Mapping

The flags determines how the linker and system loader handle the section. A section can only be of one type, one content, but can have a combination of memory flags set.

For now, all NT/OS2 objects and images will be of type **COFF_SCN_TYPE_REGULAR** except for those sections that want 16-bit offset addressing. These sections will be of type **COFF_SCN_TYPE_GROUPED**.

Section grouping is controlled by using a colon (:) in the section name. For example, if you have four objects each containing sections by the name of .DATA, .DATA:1, and .DATA:2, which all have the SAME FLAGS, then the linker will only create one section called .DATA which is a combination of all the sections but grouped in the following order:

```
Raw data for section .DATA
┌────────────────┐
│Object 1 DATA   │
│Object 2 DATA   │
│Object 3 DATA   │
│Object 4 DATA   │
│Object 1 DATA:1│
│Object 2 DATA:1│
│Object 3 DATA:1│
│Object 4 DATA:1│
│Object 1 DATA:2│
│Object 2 DATA:2│
│Object 3 DATA:2│
│Object 4 DATA:2│
└────────────────┘
```

Further more, the linker performs grouping within a file. If a file contains multiple sections with the same group name, the linker will group all raw data with the same group name within the file together. A good example of this would be a library with many members each containing a .DATA:1 group. The linker will combine all .DATA:1 raw data extracted from the library together before it combines groups of the same name from other libraries.

If a sections name is 8 characters (without the colon), then the linker will not allow it to contain groups.

**Characteristics Flags**:

**COFF_SCN_TYPE_REGULAR —**.

**COFF_SCN_TYPE_DUMMY —**.

**COFF_SCN_TYPE_NO_LOAD —**.

**COFF_SCN_TYPE_GROUPED —**Used for 16-bit offset code.

**COFF_SCN_TYPE_NO_PAD** —Specifies if section should not be padded to next boundary before being combined with other like section.

**COFF_SCN_TYPE_COPY** —Reserved.

**COFF_SCN_CNT_CODE** —Section contains code.

**COFF_SCN_CNT_INITIALIZED_DATA** —Section contains initialized data.

**COFF_SCN_CNT_UNINITIALIZED_DATA** —Section contains uninitialized data.

**COFF_SCN_CNT_OTHER** —Reserved.

**COFF_SCN_CNT_INFO** —Section contains comments or some other type of information.

A comment section can contain any type of information and can include relocations for this information. The first two long words of the raw data are reserved and are defined as InfoType and InfoVersion.

**InfoType Flags**:

**COFF_SCN_INFO_UNKNOWN** —Indicates unknown information.

**COFF_SCN_INFO_DIRECTIVE** —Indicates raw data contains linker directives such as entrypoint, full/partial/no debugging, etc. The compiler can set linker options by use of these directives. Usually the sections is also marked as discardable so this information doesn't become part of the image. InfoVersion is the linker version required to understand these directives. The current linker must have this version number or greater. The next long word is the number of directives being set, followed by the directives themselves (to be defined later). If the linker finds more than one directive of the same type (ie, two entrypoints) the linker will generated a warning and will use the first directive found.

**COFF_SCN_INFO_COMPILER** —Indicates raw data contains compiler information such as compiler type (i.e., C, Pascal, Fortran) and flags used. InfoVersion indicates the compiler version.

**COFF_SCN_INFO_CODEVIEW** —Indicates raw data contains CodeView information, and InfoVersion can either be the compiler or debugger version (to be determined later).

**COFF_SCN_CNT_OVERLAY** —Section contains an overlay.

**COFF_SCN_CNT_DISCARD** —Section contents will not become part of image. Directives to the linker will usually be marked discardable (ie, entrypoint defined by compiler).

**COFF_SCN_MEM_NOT_CACHED** —Section is not cachable.

**COFF_SCN_MEM_NOT_PAGED** —Section is not pageable.

**COFF_SCN_MEM_SHARED** —Section is shareable.

**COFF_SCN_MEM_EXECUTE** —Section is executable.

**COFF_SCN_MEM_READ** —Section is readable.

**COFF_SCN_MEM_WRITE** —Section is writeable.

### 2.3.5 Coff Relocation Entry

The relocation entries size and format is that of standard COFF.

```
typedef struct _RELOCATION_ENTRY {
    ULONG  VirtualAddress;
    ULONG  SymbolTableIndex;
    USHORT Type;
} RELOCATION_ENTRY, *PRELOCATION_ENTRY;
```

<u>RELOCATION_ENTRY Structure:</u>

*VirtualAddress* —Indicates the virtual address (position) in the section to be relocated.

*SymbolTableIndex* —Indicates the symbol table index (zero based) of the item that is referenced.

*Type* —Indicates the relocation type. Relocation types are defined later in this document.

### 2.3.6 Coff Linenumber Entry

The linenumber entries size and format is that of standard COFF.

```
typedef struct _LINENUMBER_ENTRY {
        union {
                ULONG  SymbolTableIndex;
                ULONG  VirtualAddress;
        }
        USHORT Linenumber;
} LINENUMBER_ENTRY, *PLINENUMBER_ENTRY;
```

LINENUMBER_ENTRY Structure:

*SymbolTableIndex* —If Linenumber is zero, indicates the symbol table index (zero based) of the function name.

*VirtualAddress* —If Linenumber is not zero, indicates virtual address of line number.

*Linenumber* —Indicates the line number relative to the start of the function.

## 2.3.7 Coff Symbol Table Entry

The symbol table entry size and format is that of standard COFF.

```
typedef struct _SYMBOL_TABLE_ENTRY {
        UCHAR    Name[8];
        ULONG    Value;
        SHORT    SectionNumber;
        USHORT   Type;
        CHAR     StorageClass;
        CHAR     NumberOfAuxiliaryEntries;
} SYMBOL_TABLE_ENTRY, *PSYMBOL_TABLE_ENTRY;
```

SYMBOL_TABLE_ENTRY Structure:

*Name* —Symbol name. If the first four bytes are zero, then the last 4 bytes are a pointer to the symbol in the string table. The pointer technique is used if the symbol is longer than 8 bytes.

*Value* —Symbols value dependent on section number, storage class, and type.

*SectionNumber* —The section number the symbol is defined in.

**SectionNumber meaning**:

**COFF_SYM_DEBUG** —Indicates value represents special symbolic debug information.

**COFF_SYM_ABSOLUTE** —Indicates value is an absolute value.

**COFF_SYM_UNDEFINED** —Indicates that value is used as common.

**COFF_SYM_DEFINED** —Indicates that the symbol is defined.

*Type* —Symbolic type.

**Type flags**:

**COFF_SYM_TYPE_NULL** —Indicates no type.

**COFF_SYM_TYPE_VOID** —Indicates type void.

**COFF_SYM_TYPE_CHAR** —Indicates type character.

**COFF_SYM_TYPE_SHORT** —Indicates type short integer.

**COFF_SYM_TYPE_INT** —Indicates type integer.

**COFF_SYM_TYPE_LONG** —Indicates type long integer.

**COFF_SYM_TYPE_FLOAT** —Indicates type floating point.

**COFF_SYM_TYPE_DOUBLE** —Indicates type double word.

**COFF_SYM_TYPE_STRUCT** —Indicates type structure.

**COFF_SYM_TYPE_UNION** —Indicates type union.

**COFF_SYM_TYPE_ENUM** —Indicates type enumeration.

**COFF_SYM_TYPE_MOE** —Indicates type member of enumeration.

**COFF_SYM_TYPE_UCHAR** —Indicates type unsigned character.

**COFF_SYM_TYPE_USHORT** —Indicates type unsigned short integer.

**COFF_SYM_TYPE_TYPE_UINT** —Indicates type unsigned integer.

**COFF_SYM_TYPE_ULONG** —Indicates type unsigned long integer.

*StorageClass* —Storage class of the symbol.

**StorageClass flags**:

**COFF_SYM_CLASS_EXTERNAL**

**COFF_SYM_CLASS_DLL_EXTERNAL**

**COFF_SYM_CLASS_AUTOMATIC**

**COFF_SYM_CLASS_REGISTER**

**COFF_SYM_CLASS_LABEL**

**COFF_SYM_CLASS_UNDEFINED_LABEL**

**COFF_SYM_CLASS_STATIC**

**COFF_SYM_CLASS_UNDEFINED_STATIC**

**COFF_SYM_CLASS_MEMBER_OF_STRUCT**

**COFF_SYM_CLASS_ARGUMENT**

**COFF_SYM_CLASS_STRUCT_TAG**

**COFF_SYM_CLASS_MEMBER_OF_UNION**

**COFF_SYM_CLASS_UNION_TAG**

**COFF_SYM_CLASS_TYPE_DEFINTION**

**COFF_SYM_CLASS_ENUM_TAG**

**COFF_SYM_CLASS_MEMBER_OF_ENUM**

**COFF_SYM_CLASS_REGISTER_PARAM**

**COFF_SYM_CLASS_BIT_FIELD**

**COFF_SYM_CLASS_BLOCK**

**COFF_SYM_CLASS_FUNCTION**

**COFF_SYM_CLASS_END_OF_STRUCT**

**COFF_SYM_CLASS_FILE**

**COFF_SYM_CLASS_SECTION**

*NumberOfAuxiliaryEntries* —Number of auxiliary entries that further define this symbol.

### 2.3.8 Coff Auxiliary Symbol Table Entry

In general, auxiliary entries either implement a linked list structure within the symbol table that is used for efficient access of the symbol table data by both the linker and debugger, or contain debug/relocation information that is outside the scope of the symbol table entry structure. The following auxiliary entries are defined:

- Filename - This is the first auxiliary entry in the symbol table. The contents of the auxiliary entry is either the filename (if the name is 14 characters or less), or a pointer to the string table where larger filenames are placed. Filename may contain a path.

- Section Names - This auxiliary entry follows the symbol entry for a section name. It contains the section length, the number of relocation entries for the section, and the number of line number entries for the section. This information can also be found in the section header, but by placing the information in the auxiliary entry, the debugger can obtain all needed information directly from the symbol table.

- Tagname - To be defined.

- Function - To be defined. Will probably contain prototype information.

- Block - Include special entries such as .bb (begin block), .eb (end block), .bf (begin function), .ef (end function) and .eos (end of structure).

- Array

#### 2.3.8.1 Coff Symbol Table Ordering

Because of symbolic debugging requirments, the order of symbols in the symbol table is very important. Symbols appear in the following sequence:

```
+----------------------+
| .file filename1      |
+----------------------+
| .define function1    |
+----------------------+
| .define local var1   |
| for function1        |
+----------------------+
| ...                  |
+----------------------+
| .define local varN   |
| for function1        |
+----------------------+
| .begin function      |
+----------------------+
| .block begin         |
+----------------------+
| ...                  |
+----------------------+
| .end block           |
+----------------------+
| .end function        |
+----------------------+
| statics              |
+----------------------+
| ...                  |
+----------------------+
| .file filename2      |
+----------------------+
| .define function1    |
+----------------------+
| .define local var1   |
| for function2        |
+----------------------+
| ...                  |
| ...                  |
+----------------------+
| statics              |
+----------------------+
| ...                  |
+----------------------+
│ defined global      │
│ symbols             │
├─────────────────────┤
│ undefined global    │
│ symbols             │
└─────────────────────┘
```

### 2.3.9 Coff String Table

The string table is the final component of the symbolic information. If in a symbol entry, the first four characters of the symbol's name are NULL, then the last four characters represent an offset (relative to the start of the string table) into the string table where the symbol's name is stored. Symbol names are NULL-terminated, thus the symbol's name can be any length.

The first four bytes in the string table represent a long value that specifies the number of bytes in the string table. An empty string table has a length field, but the value stored there is 0.

Internal symbols generated by compilers should try to be 8 characters or less, for these are the most efficent and require the less space.

### 2.3.10 Overlays

- o To be defined

### 2.3.11 Common Areas

Common areas are defined by the symbol record containing a non-zero value, and a zero (undefined) section number. In this case, the value is the size (number of bytes) of the symbol. The linker merges symbols of the same name and allocates the largest required space in a section called .common with content of **COFF_SCN_TYPE_UNINITIALIZED_DATA**.

### 2.3.12 16-bit Offset Definition

When sections have the SECTION_TYPE_GROUP flag set, the linker combines sections with the same name but different content flags into one section. The combined section must be 64K or less, otherwise the linker will generate an error. A special symbol will be defined by the linker that will be the address of the middle of the section, thus signed 16-bit displacements can be used by compilers. The special symbol defined by the linker will be that of the section name but the '.' (period) will be replaced with a ';' (semi-colon).

It hasn't been determined how grouping of sections with different memory flags occur. In the worst case, they must be all of one kind, probably R/W.

## 3. Fixups

- o Fixups will be performed in user mode. Thus, no code is required to verify fixups are valid (in the event an image has been tampered with).

- If the image is mapped at its specified based address, then the only runtime fixups required are those for DLL's. If the image is not mapped at the specified base address, then the fixups have to be re-applied.
- The linker will generate thunks for calls to DLL's, thus the fixups are to read/write data, not to code. Thus no Icache flushes are necessary.
- The linker will have a switch to indicate if fixups should occur as they are needed, or for a whole DLL at a time.

## 3.1 Based Relocations

The base relocations are used to re-apply fixups when an image's based address is unavailable at load time. The structure of a based entry follows:

```
typedef struct _BASED_RELOCATION_ENTRY {
        ULONG  VirtualAddress;
        ULONG  Value;
        USHORT Type;
} BASED_RELOCATION_ENTRY, *PBASED_RELOCATION_ENTRY;
```

<u>BASED_RELOCATION_ENTRY Structure:</u>

*VirtualAddress* —Indicates the virtual address (position) in the image to be relocated.

*Value* —Indicates the value of the item that is referenced. This value plus the new base should replace the word located at the virtual address.

*Type* —Indicates the relocation type. Relocation types are defined later in this document.

## 3.2 Relocation Types

### 3.2.1 I860 Relocation Types

- COFF_REL_I860_ABSOLUTE

  This relocation is ignored.

- COFF_REL_I860_DIR32

```
*(long *)Location += Addr
```

- COFF_REL_I860_PAIR

  Defines PairAddr.

- COFF_REL_I860_HIGH

  ```
  *(short *)Location = ((Addr + PairAddr) >> 16)
  ```

- COFF_REL_I860_LOW0

  ```
  *(short *)Location += (short)Addr
  ```

- COFF_REL_I860_LOW1

  ```
  *(short *)Location += (short)Aligned(Addr, 2)
  ```

- COFF_REL_I860_LOW2

  ```
  *(short *)Location += (short)Aligned(Addr, 4)
  ```

- COFF_REL_I860_LOW3

  ```
  *(short *)Location += (short)Aligned(Addr, 8)
  ```

- COFF_REL_I860_LOW4

  ```
  *(short *)Location += (short)Aligned(Addr, 16)
  ```

- COFF_REL_I860_SPLIT0

  ```
  T1 = *(long *)Location
  T2 = (((T1 >> 5) & 0xf800) | (T1 & 0x7ff)) + Addr
  T2 = ((T2 << 5) & 0x1f0000) | (T2 & 0x7ff)
  *(long *)Location = T2 | (T1 & (~0x1f07ff))
  ```

- COFF_REL_I860_SPLIT1

  ```
  T1 = *(long *)Location
  T2 = (((T1 >> 5) & 0xf800) | (T1 & 0x7fe)) + Aligned(Addr,2)
  T2 = ((T2 << 5) & 0x1f0000) | (T2 & 0x7fe)
  *(long *)Location = T2 | (T1 & (~0x1f07fe))
  ```

- COFF_REL_I860_SPLIT2

```
T1  = *(long *)Location
T2 = (((T1 >> 5) & 0xf800) | (T1 & 0x7fc)) + Aligned(Addr, 4)
T2 = ((T2 << 5) & 0x1f0000) | (T2 & 0x7fc)
*(long *)Location = T2 | (T1 & (~0x1f07fc))
```

- COFF_REL_I860_HIGHADJ

```
*(short *)Location = ((Addr + rel1.r_symndx) >> 16)
if ((Addr + rel1.r_symndx) & 0x8000)
   *(short *)Location += 1
```

- COFF_REL_I860_BRADDR

```
Addr = Addr - ((VirtAddr - PhysAddr) + 4 + VirtAddr
if ((Addr >= 0x4000000L) || (Addr < -0x4000000L))
   " Too Far "
```

I'll explain the previous relocation types by sample i860 code.

```
orh  h%foo,r0,r31            // COFF_REL_I860_HIGH
or   l%foo,r31,r31           // COFF_REL_I860_LOW0
ld.l 0(r31),r16
```

The first 2 instructions moves the address of the memory location labeled foo into r31. The COFF_REL_I860_HIGH type instructs the linker to extract the upper 16 bit of the address of foo for use as immediate operand in the orh instruction. Similarly, the COFF_REL_I860_LOW0 type instructs the linker to extract the lower 16 bit of the address of foo for use as immediate operand in the or instruction. The final ld.l loads the memory location referenced by r31 into r16.

Alternatively, you can use

```
orh       ha%foo,r0,r3   // COFF_REL_I860_HIGHADJ, PAIR
ld.l      l%foo(r31),r16 // COFF_REL_I860_LOW0
```

to load foo into r16. The COFF_REL_I860_HIGHADJ type behaves like the COFF_REL_I860_HIGH type except that it adds 1 to the extracted upper 16 bit if bit 15 of the address value is set. This adjustment is needed because load/store arithmetic instructions sign-extend the 16-bit immediate operand. If you used

```
orh       h%foo,r0,r31   // COFF_REL_I860_HIGH
ld.l      l%foo(r31),r16 // COFF_REL_I860_LOW0
```

you will load from the wrong address when bit 15 of foo is set. Immediate operands are 0-extended in logical instructions.

```
     orh       ha%foo,r0,r31  // COFF_REL_I860_HIGHADJ, PAIR
     ld.b      l%foo(r31),r16 // COFF_REL_I860_LOW0
     ld.s      l%foo(r31),r16 // COFF_REL_I860_LOW1
     ld.l      l%foo(r31),r16 // COFF_REL_I860_LOW2

     orh       ha%foof,r0,r31 // COFF_REL_I860_HIGHADJ, PAIR
     fld.l     l%foof(r31),f16// COFF_REL_I860_LOW2
     fld.d     l%foof(r31),f16// COFF_REL_I860_LOW3
     fld.q     l%foof(r31),f16// COFF_REL_I860_LOW4
```

The variaous COFF_REL_I860_LOW types are used to extract the lower 16 bits of a constant or and address label. The linker verifies alignment of the immediate offsets (Intel i860 Programmer Reference Manual section 5.2 programming notes) because the lower bits of the immediate are used to encode the operand length. See appendix B of the Intel i860 Programmers Reference Manual for the instruction format.

COFF_REL_I860_LOW1 verifies alignment of the immediate to 2 byte boundary.
COFF_REL_I860_LOW2 verifies alignment of the immediate to 4 byte boundary.
COFF_REL_I860_LOW3 verifies alignment of the immediate to 8 byte boundary.
COFF_REL_I860_LOW4 verifies alignment of the immediate to 16 byte boundary.

```
     orh       ha%foo,r0,r31  // COFF_REL_I860_HIGHADJ, PAIR
     st.b      r16,l%foo(r31) // COFF_REL_I860_SPLIT0
     st.s      r16,l%foo(r31) // COFF_REL_I860_SPLIT1
     st.l      r16,l%foo(r31) // COFF_REL_I860_SPLIT2
```

The COFF_REL_I860_SPLIT types are used by the st instruction (fst uses the COFF_REL_I860_LOW fixups). They verify the alignment of the immediate as well as split the immediate over bit 20..16 and 10..0 of the instruction. The alignment is needed because bit 0 and bit 28 are used to encode operand length.

COFF_REL_I860_SPLIT1 verifies alignment of immediate to 2 byte boundary.
COFF_REL_I860_SPLIT2 verifies alignment of immediate to 4 byte boundary.

```
     br        foo
     nop
foo: nop                      // COFF_REL_I860_BRADDR
```

The COFF_REL_I860_BRADDR type is used to fixup a br to an address label. The linker computes the offset of the target label relative to the current PC + 4.

### 3.2.2 386 Relocation Types

- COFF_REL_I386_ABSOLUTE
- COFF_REL_I386_DIR16
- COFF_REL_I386_REL16
- COFF_REL_I386_DIR32
- COFF_REL_I386_REL32

## 3.3 DLL Support

- An executable image which is a DLL will:
  - Have an export section which contains the ordinals, function names, and function address of each exported routine.
  - May contain init code if AddressOfEntryPoint != 0.
- An executable image which uses a DLL will
  - Have a Dll Descriptor table for each DLL used. These tables will be grouped together and the optional header will contain the address of the first table.
  - Thunks for the DLL that will be snaped at load time.

### 3.3.1 Thunks

The best way to describe thunks is show an example. The following example is i860 code.

Suppose we had the following Definition file:

```
GetVersion=DosCalls.GetVersion
GetMachineMode=DosCalls.GetMachineMode
GetMode=VioCalls.GetMode
Foo=DosCalls.128
```

and the following user code:

```
call GetVersion
call GetMode
call GetMachineMode
call Foo
```

The image would end up contain the following code:

```
call thunk1
call thunk2
call thunk3
call thunk4

thunk1:
    br DosCallsThunkRoutine
    ld.c fir,r31
    .word relative address of GetVersionThunkData


thunk2:
    br VioCallsThunkRoutine
    ld.c fir,r31
    .word relative address of GetModeThunkData


thunk3:
    br DosCallsThunkRoutine
    ld.c fir,r31
    .word relative address of GetMachineModeThunkData


thunk4:
    br DosCallsThunkRoutine
    ld.c fir r31
    .word relative address of Ordinal128ThunkData


DosCallsThunkRoutine:
    ld.l 0(r31),r30
    add r30,r31,r30
    ld.l 0(r30), r29
    bri r29
    nop

VioCallsThunkRoutine:
    ld.l 4(r31),r30
    add r30,r31,r30
    ld.l 0(r30), r29
    bri r29
    nop
```

Notice that DosCallsThunkRoutine and VioCallsThunkRoutine are identical. The reason for this is purely for debugging reasons. With different thunk routines, the user can set a breakpoint at the thunk routine for a specific DLL or a profiler could show which functions within which DLL are being called. The ideal situation would be to only generate one thunk routine if debugging isn't enabled, otherwise generate a thunk routine per DLL. However, I haven't figured out a way to do this yet, so until then, each DLL will have its own thunk routine.

Thunk data has the following format:

```
typedef struct _THUNK_DATA {
        PTHUNK_BY_NAME Function;
} THUNK_DATA, *PTHUNK_DATA;
```

THUNK_DATA Structure:

*Function* —Specifies either an ordinal number or a pointer to THUNK_BY_NAME structure. If it is an ordinal number, it will have a value less than 64K.

```
typedef struct _THUNK_BY_NAME {
        ULONG Hint;
        UCHAR  Name[1];
} THUNK_BY_NAME, *PTHUNK_BY_NAME;
```

THUNK_BY_NAME Structure:

*Hint* —A hint value that can be used to reference into the ExportNames in the EXPORT_SECTION_DATA.

*Name* —The functions name.

Thus by example, we have:

```
DosCallsThunkData:

GetVersionThunkData:
     .word pointer to hint & "GetVersion"


GetMachineModeThunkData
     .word pointer to hint & "GetMachineMode"


Ordinal128ThunkData:
     .word 128


VioCallsThunkData:

GetModeThunkData:
     .word pointer to hint & "GetMode"
```

The DLL descriptor is defined as:

typedef struct **_DLL_DESCRIPTOR** {
    **ULONG** *Characteristics;*
    **PUCHAR** *Name;*
    **PVOID** *FirstThunk;*
} **DLL_DESCRIPTOR**, ***PDLL_DESCRIPTOR**;

DLL_DESCRIPTOR Structure:

*Characteristics* **—**TBD.

*Name* **—**A pointer to the name of the DLL.

*FirstThunk* **—**A pointer to the first thunk for this DLL.

The linker places all DLL descriptors contigously in the image file. An empty DLL descriptor (both fields are zero) is appended to the list of DLL descriptors.The PointerToDLLTable in the optional headers points to the first DLL descriptor.

The purpose of the DLL descriptor is that once a snap occurs, it is possible to snap all thunks for the DLL at once. The linker places all the thunks for a particular DLL contiguously. It also appends an additional  thunk data record to the list. This record will have both function_ordinal and function_name set to zero. This signifies the end of the DLL thunk data.

```
Thus, by example we have:

   ┌──────────────────────┐
   │characteristics       │
   ├──────────────────────┤
   │pointer to "VioCalls" │
   ├──────────────────────┤
┌──┤pointer to first thunk│
│  ├──────────────────────┤
│  │characteristics       │
│  ├──────────────────────┤
│  │pointer to "DosCalls" │
│  ├──────────────────────┤
│┌─┤pointer to first thunk│
││ ├──────────────────────┤
││ │0                     │
││ ├──────────────────────┤
││ │0                     │
││ ├──────────────────────┤
││ │0                     │
││ ├──────────────────────┤
││ │                      │
││ │0                       │
││ └────────────────────────┘
││
││  THUNK DATA
││ ┌─────────────────────────────────┐
│└─┤ pointer to "GetVersion"         │
│  ├─────────────────────────────────┤
│  │ pointer to "GetMachineMode"     │
│  ├─────────────────────────────────┤
│  │ 128                             │
│  ├─────────────────────────────────┤
│  │ 0                               │
│  ├─────────────────────────────────┤
└──┤ pointer to "GetMode"            │
   ├─────────────────────────────────┤
   │ 0                               │
   └─────────────────────────────────┘
```

The linker doesn't know if a function is within a DLL and it doesn't have to. The thunk and thunk data will be extracted from a library that was created by the librarian from a definition file.

### 3.3.2 Export Section

The export section will be the first section header of an image that is flaged as a DLL. The raw data of the section has the following format:

```
typedef struct _EXPORT_SECTION_DATA {
        ULONG Characteristics;
        PSZ   DllName;
        ULONG VersionStamp;
        ULONG Base;
        ULONG NumberOfOrdinals;
        ULONG NumberOfNames;
        PVOID *AddressOfOrdinalFunction;
        PEXPORT_NAME_TABLE ExportNames;
} EXPORT_SECTION_DATA, *PEXPORT_SECTION_DATA;
```

<u>EXPORT_SECTION_DATA Structure:</u>

*Characteristics* —TBD.

*DllName* —A pointer to the name of the DLL.

*VersionStamp* —TBD.

*Base* —TBD.

*NumberOfOrdinals* —Indicates the number of ordinal functions.

*NumberOfNames* —Indicates the number of named functions.

*AddressOfOrdinalFunction* —A virtual address of the ordinal function.

*ExportNames* —A pointer to the function exported by name.

```
typedef struct _EXPORT_NAME_TABLE {
        PSZ ExportedName;
        ULONG  Ordinal;
} EXPORT_NAME_TABLE, *PEXPORT_NAME_TABLE;
```

<u>EXPORT_NAME_TABLE Structure:</u>

*DllName* —A pointer to the name of the Function.

*Ordinal* —The ordinal assigned to the function.

- Image headers, section headers, inport/export lists and debug information must all be mappable.
- Based images and DLL's.
- Sections are aligned on sector boundaries and are mapped on 64K virtual addresses.
- The only kernel memory resident structures is the information to resolve a virtual page to a disk location.

Psedo code for activating FOO.EXE.

```
Activate("FOO.EXE");


Activate (Image_Name)
{
  Handle = CreateSection(Image_Name, ..., ...);
  Image_Base = MapView(Handle, ..., ...);
  if (Image_Base->Optional_Header.PreferredImageBase != Image_Base){
     perform_local_fixups();
  }
  Load_DLL(Image_Base);
}


Load_DLL (Image_Base)
{
  If (ImageBase->Section_Header[0].Name == '.export') {
     while (Fetch_Next_DLL_Name() != NULL) {
       DLL_Handle = CreateSection(DLL_Name, ..., ...);
       DLL_Base = MapView(DLL_Handle, ..., ...);
       if (DLL_Base->Optional_Header.PreferredImageBase != DLL_Base) {
          perform_local_fixups();
       }
       Load_DLL(DLL_Base);
       if (Image_Base->Optional_Header.EntryPoint) {
         call (Image_Base->Optional_Header.EntryPoint());
       }
       perform_DLL_fixups();
}
```

**5. Resources**

Resources are used for internationalzation. For example, if all the error messages of an image are in a resource, then the object containing the resource can be replaced with a new resource object that contains the error messages in another language.

- o Bitmaps, Fonts, Icons and Strings can all be resources.
- o The resource compiler will not modify the executable images as is done today in OS/2. Instead, the resource compiler will produce either assembler or c language code that can be compiled and then linked with the retain flag set so it can be incrementally linked later.
- o Resouces will be combined into one section.
  - o The resource section will have a reserved name. Currently this name is .resrc.
  - o The section flag will be marked as **COFF_SCN_CNT_INITIALIZED_DATA**.

The current OS/2 implib program will be incorporated into the linker. It will read a definition file and produce a library which contains the thunk code for DLL entry points.

**6. CodeView Support**

CodeView information will reside in a section with content being **COFF_SCN_TYPE_INFO**. The Linker does not know about the internal structure of the CodeView information. The section can contain relocation entries for the information.

- o How duplicate debug information might be discarded hasn't been decided yet.

**6.1 Incremental Linking**

Incremental linking is used to replace specific parts of an image file. This is how you change resources.

The linker will be able to incrementally link objects provided the retain switch was used before incremental linking is desired. The linker will retain the needed relocation entries for each object that refers to a specific section. The linker replaces the old section with the new section and re-applys the fixups. NOTE: If the size of the section grows, and can't fit in the padded space left from sector aligning, it hasn't been decided if the linker will move everything or just return an error indicating full linking must occur.

Incremental linking is accomplished by linking an executable image with 1 or more objects.

### 6.2 Linker Command Line

The linker can except switches, objects, libraries, and the definition file in any order on the command line. Only one definition file can be specified. The linker processes the object files (in order) before processing the libraries.

### 6.3 Linker Switches

- o Debug:[None,Full,Partial]
- o Def:*Filename*
- o Dll
- o Map:[*Filename*]
- o Base:*Address* (64K aligned)
- o Entry:*SymbolName*
- o Force
- o Include:*SymbolName*
- o Out:*Filename*
- o Stack*:Size*
- o Version:*Number*
- o Retain=[All,*ObjectName*]:[All,*SectionName*]
- o Fixup=*DllLibraryName*:[All,1by1,None]

# 7. Librarian

- o The librarian will be imbedded into the linker or will at least share a DLL.
- o Multiple objects of the same name will NOT be allowed in the same library.
- o Multiple symbols of the same name will NOT be allowed in the same library.

The librarian has two functions:

The first function of the librarian is to simply merge files together. When the librarian builds a library, a member header is created for each file that is a member of the library. This allows removal of each individual file from a library. Any file can become a member of a library. When the files being added to a library are not COFF objects, the librarian acts like a simple file merger. You can merge an unlimited number of files together into one large file. This file will not be considered a valid library for linking purposes. A valid library to be used by the linker is created by merging only COFF objects together.

When a library contains only COFF objects, the librarian performs its second function, which is to build a symbol table for all defined external functions within the library. This symbol table is called the linker member, because it allows the linker to perform fast lookup on defined functions within the library. Once the linker member is created, only COFF objects can be added to the library.

## 7.1 Librarian Switches

- o Remove:Membername
- o Def:Filename
- o List

Membername is the name of the file. The linker member name is backslash (/).

```
 ___________________
|                   |
| !<arcg>\n         |
|___________________|
|                   |
| MEMBER HEADER     |
|      name         |
|      date         |
|      uid          |
|      gid          |
|      mode         |
|      size         |
|      '\n          |
|___________________|
|                   |
| File 1 Contents   |
|                   |
|                   |
|                   |
|___________________|
|                   |
| MEMBER HEADER     |
|      name         |
|      date         |
|      uid          |
|      gid          |
|      mode         |
|      size         |
|      '\n          |
|___________________|
|                   |
| File 2 Contents   |
|                   |
|                   |
|                   |
|___________________|
|                   |
|etc ...            |
|                   |
|___________________|
```

### 7.2.1 Library File Header

A library file always starts with the 8 characters **!<arch>\n** where \n is a newline character.

### 7.2.2 Library Member Header

The library member header size and format is that of standard COFF archive files.

```
typedef struct _MEMBER_HEADER {
        CHAR  Name[16];
        CHAR  Date[12];
        CHAR  UserID[6];
        CHAR  GroupID[6];
        CHAR  Mode[8];
        CHAR  Size[10];
        CHAR  EndHeader[2];
} MEMBER_HEADER, *PMEMBER_HEADER;
```

<u>MEMBER_HEADER Structure:</u>

*Name* —Is the file name of the member. It is terminated with a backslash (/) character, followed by spaces if needed to fill out the rest of the character array. The member name is stored this way only if the file name is less than 16 characters long. If the file name is 16 characters or more (path name is included), the the member names begins with a backslash (/) character, followed by ascii digits which are used as an offset into the long name table (described below).

*Date* —Is the members creation data as an ASCII string of decimal characters.

*UserID* —To be defined.

*GroupID* —To be defined.

*Mode* —To be defined.

*Size* —Defines the size of the member in bytes. The size can be used to find the header of the next member.

*EndHeader* —Contains the string **`\n** (grave accent followed by a newline character).

NOTE: A member header always starts on an even-byte boundary. A newline character (\n) is used for filling if the members contents ends on an odd-byte boundary.

### 7.2.3 Linker Member

If the file contains a COFF object, then a linker member is built by the librarian, and is the first member of the archive file. The linker member is sorted by member header offsets. The linker member is standard coff and is constructed in the following manner:

```
┌─────────────────────────┐
│ MEMBER HEADER           │
│      name "/       "    │
│      date               │
│      uid                │
│      gid                │
│      mode               │
│      size               │
│      '\n                │
├─────────────────────────┤
│ number of symbols       │
├─────────────────────────┤
│ member header offset    │
│ for symbol name1        │
├─────────────────────────┤
│ member header offset    │
│ for symbol name2        │
├─────────────────────────┤
│ .                       │
│ .                       │
│ .                       │
├─────────────────────────┤
│ member header offset    │
│ for symbol nameN        │
├─────────────────────────┤
│ symbol name1            │
│ symbol name2            │
│                         │
│                         │
│                         │
│ symbol nameN            │
└─────────────────────────┘
```

### 7.2.4 Secondary Linker Member

A second linker member is built by the NT librarian. This is not standard, but most existing tools should ignore the second linker member. The second linker member is sorted by symbols names. The second linker member is constructed in the following manner:

```
┌─────────────────────────┐
│ MEMBER HEADER           │
│      name "/       "    │
│      date               │
│      uid                │
│      gid                │
│      mode               │
│      size               │
│      '\n                │
├─────────────────────────┤
│ number of offsets       │
├─────────────────────────┤
│ member header 1 offset  │
├─────────────────────────┤
│ member header 2 offset  │
├─────────────────────────┤
│ .                       │
│ .                       │
│ .                       │
├─────────────────────────┤
│ number of symbols       │
├─────────────────────────┤
│ member offset index     │
│ for symbol name1        │
├─────────────────────────┤
│ member offset index     │
│ for symbol name2        │
├─────────────────────────┤
│ .                       │
│ .                       │
│ .                       │
├─────────────────────────┤
│ member offset index     │
│ for symbol nameN        │
├─────────────────────────┤
│ symbol name1            │
│ symbol name2            │
│                         │
│                         │
│                         │
│ symbol nameN            │
```

```
├────────────────────────┤
│ full member 1 filename │
│ full member 2 filename │
│                        │
│                        │
│ full member N filename │
└────────────────────────┘
```

### 7.2.5 Long Names Member

The NT linker builds a long name table if any of the file names being added to the library are longer than 15 characters. This is not standard COFF, but is part of the new System V ABI. The long name table is constructed in the following manner:

```
┌────────────────────────┐
│ MEMBER HEADER          │
│      name "//      "   │
│      date              │
│      uid               │
│      gid               │
│      mode              │
│      size              │
│      '\n               │
├────────────────────────┤
│ asciiz strings         │
└────────────────────────┘
```

## Revision History

Original Draft 1.0, November 06, 1989

Revision 1.1, January 10, 1990

Revision 1.2, Febuary 26, 1990

Revision 1.3, May 31, 1990