IBM OS/2 16/32-bit Object Module Format (OMF) and Linear eXecutable Module Format (LX)

Revision 8

June 30, 1994

Boca Programming Center
Boca Raton, Florida

Purpose of this document

THIS DOCUMENT PROVIDED BY IBM SHALL BE PROVIDED ON AN "AS IS" BASIS WITHOUT ANY WARRANTY OF ANY KIND EITHER EXPRESS OR IMPLIED. THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE EXPRESSLY DISCLAIMED.

FURTHERMORE, THIS DOCUMENTATION IS IN A PRELIMINARY FORM; IS NOT COMPLETE; HAS NOT YET BEEN TESTED, VALIDATED OR REVIEWED; MAY CONTAIN ERRORS, OMISSIONS, INACCURACIES OR THE LIKE; AND IS SUBJECT TO BEING CHANGED, REVISED OR SUPERSEDED IN WHOLE OR IN PART BY IBM. IBM DOES NOT ASSUME ANY RESPONSIBILITY TO NOTIFY ANY PARTIES, COMPANIES, USERS, AND OR OTHERS OF DEFECTS, DEFICIENCIES, CHANGES, ERRORS OR OTHER FAILINGS OR SHORTCOMING OF THE DOCUMENTATION.

RECIPIENT'S USE OF THIS DOCUMENT IS LIMITED TO RECIPIENT'S PERSONAL USE FOR THE SOLE PURPOSE OF CREATING TOOLS FOR THE OS/2[1] OPERATING SYSTEM.

[1] OS/2 is a Registered Trademark of International Business Machines Corp.

Contents

Figures

Major changes to this document

- Draft 1 = Combined information from several documents into one.
- Draft 2 = Added Comments from Lexington and Toronto.
- Draft 3 = Added the Linear Executable format (LX).
- Draft 4 = Minor corrections.
- Draft 5 = Added StackSize to LX structure.
- Revision 6 = Added IDMDLL COMENT record
- Revision 7 = Added 16-bit object record definitions
- Revision 8 = Added ITERDATA2 definition and minor corrections

Introduction

This document is intended to describe the interface that is used by language translators and generators as their intermediate output to the linker for the 32-bit OS/2 operating system. The linker will generate the executable module that is used by the loader to invoke the .EXE and .DLL programs at execution time.

THE 16/32-BIT OBJECT MODULE FORMAT

Record Format:

All object records confirm to the following format:

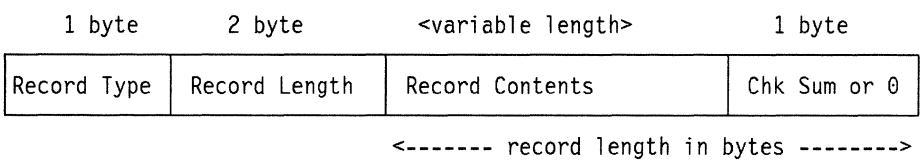


Figure 1. Standard object module record format.

The **Record Type** field is a 1-byte field containing the hexadecimal number that identifies the type of object record. The format is determined by the least significant bit of the Record Type field. Note that this does not govern Use32/Use16 segment attributes; it simply specifies the size of certain numeric fields within the record. An odd Record Type indicates that 32-bit values are present. An even Record Type indicates that those fields contain 16-bit values. The fields affected are described with each record.

An entire record occupies **Record Length** + 3 bytes. The record length does not include the count for the record type and record length fields. Unless otherwise noted within the record definition, the record length should not exceed 1024 bytes.

The **Record Contents** are determined by the record type.

The **Chk Sum** field is a 1-byte field that contains the negative sum (modulo 256) of all other bytes in the record. The byte sum over the entire record, ignoring overflow, is zero.

NOTES:

LINK386 ignores the value of the Chk Sum byte.

Frequent Object Record Subfields

The contents of each record are determined by the record type, but certain subfields appear frequently; the format of such fields is described next.

Names

Name strings are encoded as an 8-bit unsigned count followed by a string of "count" characters. The character set is usually some ASCII subset. A null name is specified by a single byte of 0 (indicating a string of length zero).

Indexed References

Certain items are ordered by occurrence, and referenced by index (starting index is 1). Index fields can contain 0, indicating not-present, or values from 1 through 7FFF. The index is encoded as 1 or 2 bytes.

If the index number is in the range 0-7H, the high-order bit (bit 7) is 0 and the low-order bits contain the index number, so the field is only 1 byte long. If the index number is in the range 80-7FFFH, the field is 2 bytes long. The high-order bit of the first byte in the field is set to 1, and the high-order byte of the index number which must be in the range (0-7FH) fits in the remaining 7 bits. The low-order byte of the index number is specified in the second byte of the field. A 16-bit value is obtained as follows:

```
if (first_byte & 0x80)
   index_word  = (first_byte & 7F) * 0x100  +  second_byte;
else
   index_word = first_byte
```

Type indices

The type index is treated as an index field when a record is parsed (occupies one or two bytes, occurs in PUBDEF, COMDEF, EXTDEF records). They are encoded as described under indexed references.

NOTE: At present, no type checking is done by the linker. If any link-time semantics are defined, that information will be recorded somewhere within this document.

Ordered Collections

Certain records and record groups are ordered; the ordering is obtained from the order of the record types within the file together with the ordering of repeated fields within these records. Such ordered collections are referenced by index, counting from 1 (index 0 indicates unknown or decline-to-state).

For example, there may be many LNAMES records within a module and each of those records may contain many names. The names are indexed starting at 1 for the first name in the first LNAMES record encountered while reading the file, 2 for the second name in the first record, etc., and the highest index for the last name in the last LNAMES record encountered.

The ordered collections are:

- NAMES: ordered by LNAMES record and names within each. Referenced as a Name Index.
- LOGICAL SEGMENTS: ordered by SEGDEF records in file. Referenced as a Segment Index.
- GROUPS: ordered by GRPDEF of records in file. Referenced as a Group Index.
- EXTERNAL SYMBOLS: ordered by EXTDEF and COMDEF records and symbols within each. Referenced as an External Index (in FIXUPs).

Numeric 2 and 4 byte fields

Words and double words (16 and 32 bit quantities) are stored in Intel byte order (lowest address is least significant).

Certain records, notably SEGDEF, PUBDEF, LINNUM, LEDATA, LIDATA, FIXUPP and MODEND, contain size, offset, and displacement values which may be 32 bit quantities for Use32 segments. The encoding is as follows.

- When the least significant bit of the record type byte is set (ie record type is an odd number), the numeric fields are 4 bytes.
- When the least significant bit of the record type byte is clear, the fields occupy 2 bytes (16 bit Object Module Format). The values are zero-extended when applied to Use32 segments.

See the description of SEGDEF records for an explanation of Use16/Use32 segments.

Order of records

The record order is chosen so that bind/link passes through an object module are minimized. This differs from the previous less specific ordering in that all symbolic information (in particular, all export and public symbols) must occur at the start of the object module. This order is recommended but not mandatory.

Identifier record(s):

Must be the first record.

- THEADR or LHEADR

Records processed by Link Pass one:

May occur in any order but must precede the Link pass separator if it is present.

- COMENT class AF providing name of Identifier Manipulator Dynamic Link Library (should be near the beginning of the file)
- COMENT identifying object format and extensions
- COMENT any, other than link pass separator comment
- LNAMES providing ordered name list
- SEGDEF providing ordered list of program segments
- GRPDEF providing ordered list of logical segments
- TYPDEF (no longer used)
- ALIAS records
- PUBDEF locating and naming public symbols
- LPUBDEF locating and naming private symbols.
- COMDEF, EXTDEF, LCOMDEF, LEXTDEF records

  This group of records is indexed together, so External Index fields in FIXUPP records may refer to any of the record types listed.
- COMDAT records

Link pass separator (optional):

- COMENT class A2 indicating that pass 1 of the linker is complete.

  When this record is encountered, LINK immediately starts Pass 2; no records after this comment are read in Pass 1. All the above listed records must come before this comment record. For greater linking speed, all LIDATA, LEDATA, FIXUPP and LINNUM records should come after the A2 comment record, but this is not required.

In LINK, Pass 2 begins again at the start of the object module, so LIDATA records, etc., are processed in Pass 2 no matter where they are placed in the object module.

Records ignored by link pass one and processed by link pass two:

May come before or after link pass two:

- LIDATA or LEDATA records followed by applicable FIXUPP records.
- FIXUPPs containing THREADs only.
- BAKPAT and NBAKPAT fixupps.
- LINNUM and LINSYM providing line number to program code or data association.

Terminator

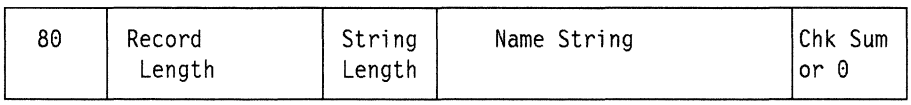
- MODEND indicating end of module with optional start address.

Object Record Types

80H THEADR Translator Header Record

Description:

The THEADR record contains the name of the object module. This name identifies an object module within an object library or in messages produced by the linker.



1 byte	2 byte	1 byte	< variable length >	1 byte
80	Record Length	String Length	Name String	Chk Sum or 0

Figure 2. THEADR record type definition

The **String Length** byte gives the number of characters in the name string; the name string itself is ASCII. This name is usually that of the source program (if supplied by the language translator), or may be specified directly by the programmer (e.g. TITLE pseudo-op).

This record must occur as the first object record. More than one header record is allowed (as a result of an object bind, or if source arose from multiple files as a result of include processing).

NOTES:

The name string is always present; a null name is allowed but not recommended (not much information for a debugger that way).

It is recommended that the module be generated with the full path and filename containing the source code.

The THEADR record must be the first record of the object module.

More than one header record is allowed (as a result of source from multiple files during the include process).

82H LHEADR Library Header Record

Description:

This record is very similar to the THEADR record. It is used to indicate the name of a module within a library file (which has a different organization internally than an object module).

Record format:

1 byte	2 byte	1 byte	<variable length>	1 byte
82	Record Length	String Length	Name String	Chk Sum or 0

Figure 3. LHEADR record type definition

NOTES:

In LINK, THEADR and LHEADR records are handled identically.

88H COMENT Comment Record

Description:

The COMENT record contains a character string that may represent a plain text comment, a symbol meaningful to a program such as LINK or LIB, or some binary coded information that alters the linking process. The comment records are actually a group of items, classified by "comment class".

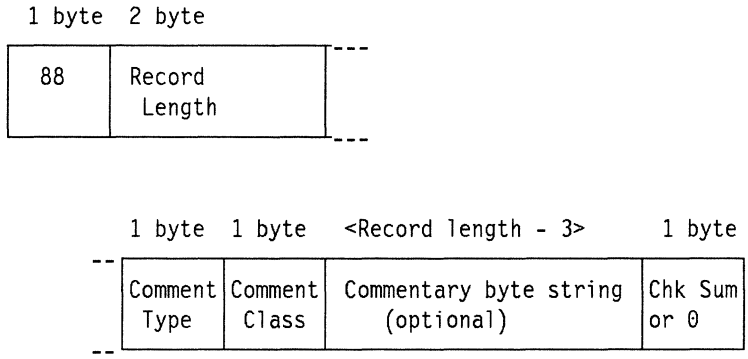
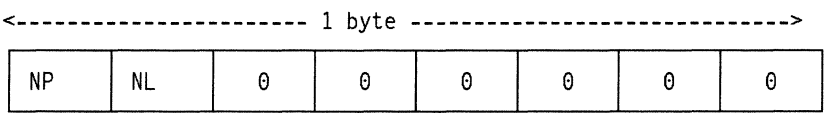


Figure 4. COMENT record type definition

Comment Type

The comment type byte is bit-significant; layout is:



where

NP is set if the comment is to be preserved by object bind utilities
NL is set if the comment is not for display by object bind utilities

Comment class and commentary byte string

The comment class is an 8-bit numeric which conveys information by its value (accompanied by a null byte string), or indicates the information to be found in the accompanying byte string. The byte string's length is determined from the record length, not by an initial count byte.

The values in use currently are the following:

0 Translator
For translator; may name the source language or translator. Recommended: translator name and version plus optimization level used for compilation be recorded here. Other compiler or assembler options can be included, although current practice seems to be to place these under comment class 9D.

1 Intel copyright
Ignored by the linker.

2 through 9B Intel reserved
The values from 9C through FF are ignored by Intel products.

9C MS-DOS version -- obsolete
Ignored by linker

9D Memory Model -- ignored
Ignored by linker

9E DOSSEG
Sets the linkers DOSSEG switch. The byte string is null. This record is included in the startup module in each language library. It directs the linker to use the standardized segment ordering, according to the naming conventions documented with DOS, OS/2 and accompanying language products.

9F Library indicator
The byte string contains a library file name (without a lead count byte and without an extension). Can be over-ridden via NOD link switch.

A0 OMF extensions
This class consists of a set of records, identified by subtype (first byte of commentary string). Values supported by the OS/2 2.0 linker are

01 **IMPDEF**
Import definition record. See IMPDEF section for complete description.

02 **EXPDEF**
Export definition record. See EXPDEF section for complete description.

03 **INCDEF**
Incremental compilation record. See INCDEF section for complete description.

04 **Protected Memory Library**
Relevant to 32 bit DLL's.This comment record is inserted in the object module by the compiler when it encounters a compiler option or pragma indicating a protected DLL. The linker then sets a flag in the header of the executable file (DLL) to indicate that the DLL should be loaded in such a way that its shared code and data is protected from corruption.

When the flag is set in the EXE header, the loader loads the selector of the protected memory area into the DS while performing run-time fixups (relocations). All other DLL's and applications get the regular DGROUP selector, which doesn't allow access to the protected memory area set up by the operating system.

05 **LNKDIR**
C++ linker directives record. See LNKDIR section for complete description.

06-FF Reserved for Microsoft.

NOTE: presence of any unrecognized subtype causes LINKER to generate a fatal error.

A1 Symbolic debug information
This comment class is now used solely to indicate the version of the symbolic debug information.

The byte string will be a version number (8-bit numeric) followed by an ASCII character string indicating the style of symbol and line number (LINNUM) information. Current values are

n,'C','V' CodeView style
n,'D','X' AIX style
n,'H','L' IBM PM Debugger

A2 Link Pass
This record conveys information to the linker about the organization of the file. At present, a single sub-extension is defined. The commentary string is

01	Optional

Subclass 01 indicates the start of link pass 2 records; this may be followed by anything at all, which will be ignored by the linker (determined from the RecLength). When this comment appears, the linker can rest assured that only LEDATA, LIDATA, FIXUPP, LINNUM and the terminal MODEND records will occur after this. All other record types, plus THREAD fixups, occur before.

WARNING: It is assumed that this comment will not be present in a module whose MODEND record contains a program starting address.

A3 LIBMOD indicator
Library module comment record. Ignored by LINK386.

A4 EXESTR indicator
Executable Module Identification String

A commentary string specifying a string to be placed in the executable module, but which is not loaded with the load module.

A6
INCERR

Incremental compilation error. See INCERR section for a complete description.

A7
NOPAD

No segment padding. Ignored by LINK386.

A8 WKEXT
Weak Extern record. See WKEXT section for a complete description.

A9 LZEXT
Lazy Extern record. Ignored by LINK386.

AA PHARLAP
PharLap Format record. Ignored by LINK386.

AF IDMDLL indicator
Identifier Manipulator Dynamic Link Library. See IDMDLL section for a complete description

B2H-BFH
Unused

C0H-FFH
Reserved for user-defined comment classes.

Notes:

A COMENT record can appear almost anywhere in an object module. Only two restrictions apply:

- A COMENT record cannot be placed between a FIXUPP record and the LEDATA or LIDATA record to which it refers.
- A COMENT record can not be the first or last record in an object module. (The first record must always be a THEADR record and the last must always be a MODEND).

88H IMPDEF Import Definition Record (comment class A0, subtype 01)

Description:

This record describes the imported names for a module.

Record format:

One import symbol is described; the subrecord format is

1	1	<variable>	<variable>	2or<var> (bytes)
01	Ord Flag	Internal Name	Module Name	Entry Ident

Figure 5. IMPDEF record type definition

where:

01 identifies the subtype as an IMPDEF

OrdFlag is a byte; if zero the import is identified by name. If nonzero, it is identified by ordinal. Determines the form of the EntryIdent field.

InternalName in <count, char> string format and is the name used within this module for the import symbol. This name will occur again in an EXTDEF record.

ModuleName in <count, char> string format and is the name of the module which supplies an export symbol matching this import.

EntryIdent is an ordinal or the name used by the exporting module for the symbol, depending upon the OrdFlag.

If this field is an ordinal (OrdFlag nonzero), it is a 16-bit word. If this is a name, and the first byte of the name is zero, then the exported name is the same as the import name (in the InternalName field). Otherwise, it is the imported name in <count, char> string format (as exported by ModuleName).

Notes:

IMPDEF records are created by the utility IMPLIB, which builds an "import library" from a module definition file or dynamic-link library.

88H EXPDEF Export Definition Record (comment class A0, subtype 02)

Description:

This record describes the exported names for a module.

Record format:

One exported entry point is described; the subrecord format is

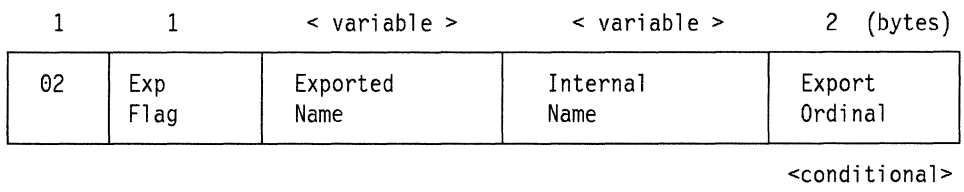
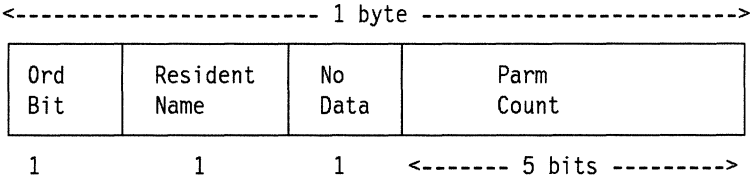


Figure 6. IMPDEF record type definition

where:

02 identifies the subtype as an EXPDEF

ExpFlag is a bit-significant 8-bit field.



OrdBit Set if the item is exported by ordinal; in this case the ExportOrdinal field is present.

ResName Set if the exported name is to be kept resident by the system loader; this is an optimization for frequently used items imported by name.

NoData Set if the entry point does not use initialized data (either instanced or global).

ParmCount Number of parameter words. The ParmCount field is set to zero for all but callgates to 16-bit segments.

Exported Name in <count, char> string format. Name to be used when the entry point is imported by name.

Internal Name in <count, char> string format. If the name length is zero, the internal name is the same as the Exported Name. Otherwise, it is the name by which the entry point known within this module. This name will appear as a PUBDEF or LPUBDEF name.

ExportOrdinal present if the OrdBit is set; it is a 16-bit numeric whose value is the ordinal used (must be non-zero).

Notes:

EXPDEFs are produced by the compiler when the keyword **_export** is used in a source file. LINK386 limits the ExportOrdinal value to 16384(16K) or lower.

88H INCDEF Incremental Compilation Record (comment class A0, subtype 03)

Description:

This record is used for incremental compilation. Every FIXUPP and LINNUM record folliwing an INCDEF record will adjust all external index values and line number values by the appropriate delta. The deltas are cumulative if there is more than one INCDEF per module.

Record format:

The subrecord format is

1	2	2	<variable> (bytes)
03	EXTDEF delta	LINNUM delta	padding

Figure 7. INCDEF record type definition

The EXTDEF delta and LINNUM delta fields are signed.

Padding (zeros) is added by Quick C to allow for expansion of the object module during incremental compilation and linking.

Notes:

Negative deltas are allowed.

88H LNKDIR C++ Directives Record (comment class A0, subtype 05)

Description:

This record is used by the compiler to pass directives and flags to the linker.

Record format:

The subrecord format is

1	1	1	1 (bytes)
05	Bit Flags	Pseudocode Vers	CV Vers

Figure 8. LNKDIR record type definition

The format of the **Bit Flags** byte is:

8	1	1	1	1	1	1	1	1 (bits)
05	0	0	0	0	0	Run MPC	Omit CV Publics	New EXE

Figure 9. BIT FLAGS byte definition

The low-order bit, if set, indicates that LINK386 should output the new EXE format; this flag is ignored for all but linking of Pseudocode applications. (Pseudocode requires a segmented executable) .

The second low-order bit indicates that LINK386 should not output the $PUBLICS subsection of the CodeView info.

The third low-order bit indicates that MPC (Microsoft Make Pseudocode Utility) should be run.

Pseudocode Version
One byte indicating the Pseudocode interpreter version number.

CodeView Version
One byte indicating the CodeView version number.

Notes:

The presence of this record in an object module will indicate the presence of global symbols records. The linker will not emit a Publics section for those modules with this comment record and a $SYMBOLS section.

88H LIBMOD Library Module Name Record (comment class A3)

Description:

The LIBMOD comment record is used only by the LIB utility, not by LINK. It gives the name of an object module within a library, allowing LIB to preserve the library file name in the THEADR record and still identify the module names that make up the library. Since the module names is the basename of the .OBJ file that was built into the library, it may be completely different from the final library name.

Record format:

The subrecord format is

1	<variable> (bytes)
A3	Module Name

Figure 10. LIBMOD record type definition

The record contains only the ASCII string of the module name, in <coutn, char> format. The module name has no path and no extension, just the base of the module name.

Notes:

LIB adds a LIBMOD record when a .OBJ file is added to a library and strips the LIBMOD record when a .OBJ file is removed from a library, so typically this record only exists in .LIB files.

There will be one LIBMOD record in the library file for each object module that was combined to build the library.

LINK386 ignores LIBMOD coment records.

88H EXESTR Executable String Record (comment class A4)

Description:

The EXESTR comment record implements the ANSI and XENIX/UNIX features in C:

- #pragma comment(exestr, < char-sequence >)
- #ident string

Record format:

The subrecord format is

1	<variable> (bytes)
A4	arbitrary text

Figure 11. EXESTR record type definition

The linker will copy the text in the "arbitrary text" field byte for byte to the end of the executable file. The text will not be included in the program load image.

Notes:

If CodeView information is present, the text will not be at the end of the file, but somewhere before so as not to interfere with the Code View signature.

There is no limit to the number of EXESTR comment records.

88H INCERR Incremental Compilation Error (comment class A6)

Description:

This comment record will cause the linker to terminate with the fatal error saying something to the effect of "inavlid object -- error encountered during incremental compilation".

The purpose of this is for the case when an incremental compilation fails and the user tries to manually link. the object module cannot be deleted, in order to preserve the base for the next incremental compilation.

Record format:

The subrecord format is

1 (bytes)	
A6	No fields

Figure 12. INCERR record type definition

88H NOPAD No Segment Padding (comment class A6)

Description:

This comment record identifies a set of segments which are to be excluded from the padding imposed with the /PADDATA or /PADCODE options.

Record format:

The subrecord format is

1	1 or 2 (bytes)
A7	SEGDEF Index

Figure 13. NOPAD record type definition

The SEGDEF Index is the standard OMF index type od 1 or 2 bytes. It may be repeated.

Notes:

LINK386 ignores NOPAD coment records.

88H WKEXT Weak Extern Record (comment class A8)

Description:

This record marks a set of external names as "weak", and for every weak extern associates another external name to use as the default resolution.

Record format:

The subrecord format is

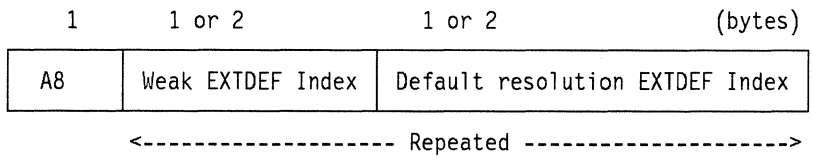


Figure 14. WEAK EXTERN record type definition

The Weak EXTDEF Index field is the 1 or 2 byte index to the EXTDEF of the extern which is weak.

The Default Resolution EXTDEF Index is the 1 or 2 byte index to the EXTDEF of the extern that will be used to resolve the extern if no "stronger" link is found to resolve it.

Notes:

There are two ways to cancel the "weakness" of a weak extern; both result in the extern becoming a "strong" extern (the same as an EXTDEF). They are:

- if a PUBDEF for the weak extern is linked in,
- if an EXTDEF for the weak extern is found in another module (including libraries).

If the weak extern becomes strong, then it must be resolved with a matching PUBDEF, just like a regular EXTDEF. If a weak exten has not become strong by the end of the linking process, then the default resolution is used.

If two weak externs for the same symbol in different modules have differing default resolutions, LINK386 will emit a warning.

Weak externs do not query libraries for resolution; if an extern is still weak when libraries are searched, it stays weak and gets the default resolution. However, if a library module is linked in for other reasons (say, to resolve strong externs) and there are EXTDEFs for symbols that were weak, the symbols become strong.

For example, suppose there is a weak extern for "foo" with a default resolution name of "bar". If there is a PUBDEF for "foo" in some library module which would not otherwise be linked in, then the library module is not linked in, and any references to "foo" are resolved to "bar". However, if the library module is linked in for other reasons, for example to resolve references to a strong extern named "bletch", then "foo" will be resolved by the PUBDEF from the library, not to the default resolution "bar".

WKEXTs are best understood by explaining why they were added in the first place. The minimum BASIC runtime library in the past consisted of a large amount of code which was always linked in, even for the smallest program. Most of this code was never called directly by the user, but it was called indirectly from other routines in other libraries, so it had to be linked in to resolve the external references.

For instance, the floating point library was linked in even if the user's program did not use floating point, because the PRINT library routine contained calls to the floating point library for support to print floating point numbers.

The solution was to make the function calls between the libraries into weak externals, with the default resolution set to a small stub routine. If the user never used a language construct or feature that needed the additional library support, then no strong extern would be generated by the compiler and the default resolution (to the stub routine) would be used. However, if the user accessed the library's routines or used constructs that required the library's support, a strong extern would be generated by the compiler to cancel the effect of the weak extern, and the library module would be linked in. This required that the compiler know a lot about which libraries are needed for which constructs, but the resulting executable was much smaller.

88H LZEXT Lazy Extern Record (comment class A9)

Description:

This record marks a set of external names as "lazy", and for every lazy extern associates another external name to use as the default resolution.

Record format:

The subrecord format is

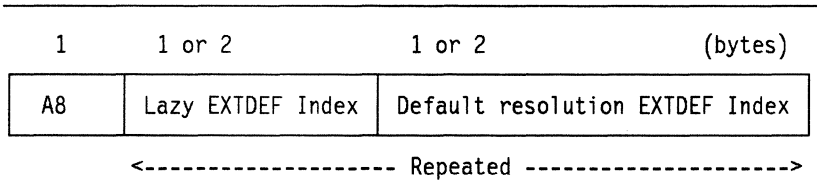


Figure 15. LAZY EXTERN record type definition

The Lazy EXTDEF Index field is the 1 or 2 byte index to the EXTDEF of the extern which is weak.

The Default Resolution EXTDEF Index is the 1 or 2 byte index to the EXTDEF of the extern that will be used to resolve the extern if no "stronger" link is found to resolve it.

Notes:

There are two ways to cancel the "laziness" of a lazy extern; both result in the extern becoming a "strong" extern (the same as an EXTDEF). They are:

- if a PUBDEF for the weak extern is linked in,
- if an EXTDEF for the weak extern is found in another module (including libraries).

If a lazy extern becomes strong, then it must be resolved with a matching PUBDEF, just like a regular EXTDEF. If a lazy extern has not become strong by the end of the linking process, then the default resolution is used.

If two weak externs for the same symbol in different modules have differing default resolutions, LINK will emit a warning.

Unlike waek externs, lazy externs do not query libraries for resolution; if an extern is still lazy when libraries are searched, it stays lazy and gets the default resolution.

LINK386 ignores LZEXT coment records.

88H IDMDLL Identifier Manipulator DLL (comment class AF)

Description:

This record provides the name and initialization parmameters of a DLL that will demangle the compiler generated mangled names. The linker will use this DLL when displaying error messages.

Record format:

The Subrecord Format is:

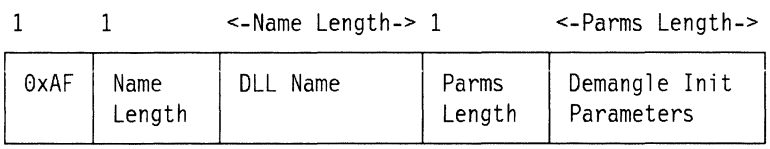


Figure 16. IDMDLL Identifier Manipulator DLL subrecord format definition

The Name Length byte gives the number of characters in the DLL Name; the DLL Name itself is ASCII.

The DLL Name is the name of the Identifier Manipulator Dynamic Link Library provided by the language. This DLL is used to demangle an internal identifier when that identifier will be displayed in an error message.

The Parms Length byte gives the number of characters in the Demangle Init Parameters; the Demangle Init Parameters itself is ASCII.

The Demangle Init Parameters provides information (to the DLL) on how internal identifiers are mangled.

The linker will not scan forward for an IDMDLL record when an identifier will be displayed. This record should occur near the beginning of the file.

IDMDLL class COMENT records are processed during pass 1 of the linker.

Notes:

Because object oriented compilers allow for two functions to have the same name but different parameters, the compiler uniquely identifies each function by changing the name of the function. This is known as mangling. An example of this would be:

```
User Prototype                Compiler Generated
                              Mangled Name
-------------------------     -----------------
void doit( int, float )       _doit__Fif
void doit( const char * )     _doit__FCPc
```

The user will usually not be aware that the compiler changed the name, so it is neccessary for the linker to demangle the compiler generated name when printing out linker error messages.

The dynamic link library (DLL) provided by an object oriented language compiler must contain two 16-bit functions which employ the pascal calling convention:

INITDEMANGLEID Receive initialization parameters specified in the IDMDLL COMENT record.

DEMANGLEID Demangles first parameter (identifier, "_add__i_ii") to appropriate prototype (i.e. "int add(int, int)") and returns result in second parameter.

The INITDEMANGLEID and DEMANGLEID entry points may be called more than once.

All functions must return true (non-zero) if the call is successful and false (zero) if the call fails. In this manner the linker can ignore whatever is returned in the second parameter of the DEMANGLEID fuction if the function returns false. When calling DEMANGLEID, the linker will pass in the address of a buffer for the second parameter, and the size of the buffer for the third parameter.

All string parameters must be length-prefixed ASCII strings except for pszPrototype, parameter 2 for DEMANGLEID (because the length might not fit in a byte). Function prototypes for these routines look like:

```
unsigned short pascal far INITDEMANGLEID(char far * psInitParms);

unsigned short pascal far DEMANGLEID(char far * psMangledName,
                                     char far * pszPrototype,
                                     unsigned long BufferLen);
```

Note: Languages may also wish to provide 32-bit functions for use by 32-bit linkers, when they become available. Function prototypes look like:

```
unsigned long _system InitDemangleID32(char * psInitParms);

unsigned long _system DemangleID32(char * psMangledName,
                                   char * pszPrototype,
                                   unsigned long BufferLen);
```

88H PharLap Format Record (comment class AA)

Description:

The OMF extension designed by PharLap is called "Easy OMF-386" and changes to the affected record types are described in this section.

Most modifications involve only a substitution of 32-bit (4-byte) fields for what were formerly 16-bit (2-byte) fields. In the two cases where the changes involve more than just a field size (in the SEGDEF and FIXUPP records), the information is mentioned in this section but complete details are given in the sections describing the specific records.

Record format:

The subrecord format is

AA	"80386"

Notes:

The AA comment record should come immediately after the sole THEADR record. Presence of the comment record indicates that the following other record types have fields that are expanded from 16-bit to 32-bit values:

SEGDEF offset field and offset field length

PUBDEF offset field

LEDATA offset field

LIDATA offset field (note that repeat count field is still 16 bits)

FIXUPP target displacement in explicit FIXUP subrecord

BLKDEF return address offset field

LINNUM offset field

MODEND target displacement field

FIXUPP records have the added Loc values of 5 and 6. See the FIXUPP section of this document for details.

SEGDEF records have added alignment values (for 4-byte alignment and 4K byte alignment) and an added optional byte at the end which contains the Use16/Use32 bit flag and access attributes (read/write/execute) for the segment. See the SEGDEF section of this document for details.

LINK386 ignores PHARLAP coment records.

8AH or 8BH MODEND Module End Record

Description:

The MODEND record denotes the end of the object module. It also indicates whether the object module contains a main routine in a program, and it can, optionally, contain a reference to a programs entry point.

Record format:

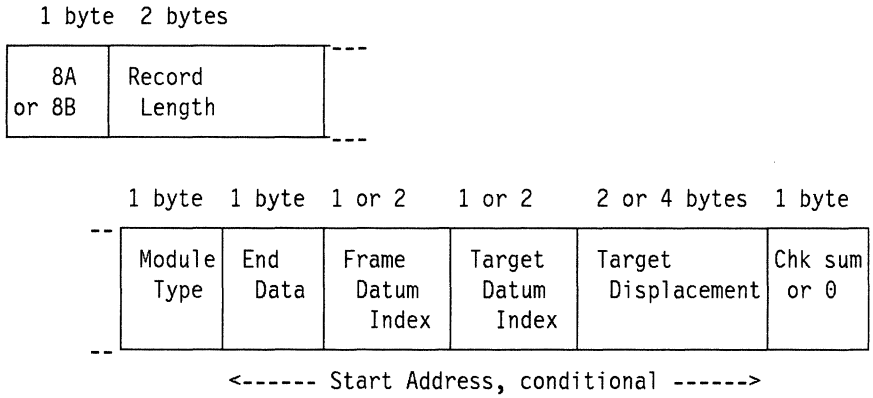
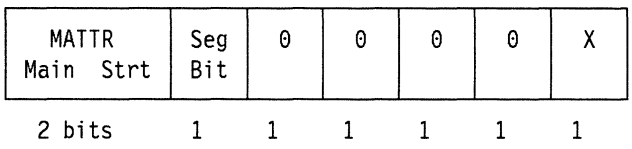


Figure 17. MODEND module end record

where:

Module Type

The module type byte is bit-significant; layout is:



where:

MATTR is a 2-bit field

Main is set if the module is a main module

Strt is set if the module contains a start address; if this bit is set, the field starting with the EndDat byte is present and specifies the start address.

SegBit Reserved. Only 0 is supported by OS/2.

X This bit should be set (as described for OMF86). However, as is the case for the OMF86 linkers, the value will be ignored.

Start Address

The Start Address subfield is present only if the Strt bit in the Module Type byte is set. Its format is identical to the FixDat, Frame Datum, Target Datum, and Target displacement in a FIXUP subrecord of a FIXUPP record. The displacement (if present) is a 4 byte field if the record type is 8BH and is a 2-byte field if the record type is 8AH. This value provides the initial contents of CS:(E)IP.

The start address must be given in th MODEND record of the root module if overlays are used.

Notes:

A MODEND record can appear only as the last record in an object module.

It is assumed that the link pass separator comment record (COMENT A2, subtype 01) will not be present in a module whose MODEND record contains a program starting address.

8CH EXTDEF External Names Definition Record

Description:

The EXTDEF record contains a list of symbolic external references -- that is, references to symbols defined in other object modules. The linker resolves external references by matching the symbols declared in EXTDEF records with symbols declared in PUBDEF records.

Record format:

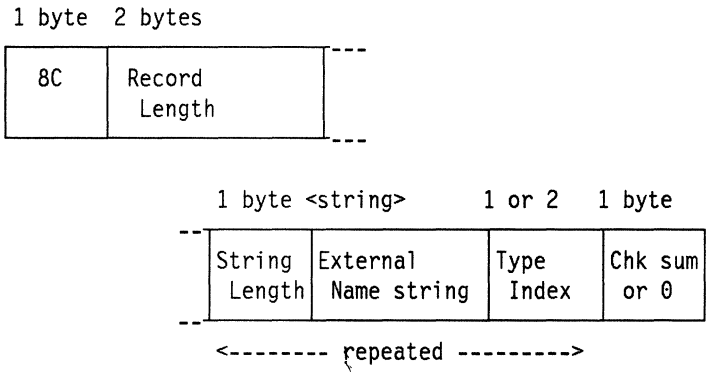


Figure 18. EXTDEF external names definition record

This record provides a list of unresolved references, identified by name and with optional associated type information. The external names are ordered by occurrence jointly with the COMDEF and LEXTDEF records and referenced by an index in other records (FIXUPPs); the name may not be null. Indices start from one.

String Length is a 1-byte field containing the length of the name field that follows it. The length of the name is restricted to 255 bytes.

The Type Index is encoded as an index field and contains debug information. No type checking is performed by the linker.

Notes:

The linker imposes a limit of 1023 external names.

Any EXTDEF records in an object module must appear before the FIXUPP records that reference them.

Resolution of an external reference is by name match (case sensitive) and symbol type match. The search first looks for a matching name, in the sequence:

1. Searches PUBDEF and COMDEF for resolution.
2. If linking a segmented executable, searches imported names (IMPDEF).
3. If this is not a DLL, then searches for an export (EXPDEF) with the same name -- a self-imported alias.
4. Searches for the symbol name among undefined symbols. If the reference is to a weak extern, then the default resolution is used. If the reference is to a strong extern, then it's an undefined external and a link error is generated.

All external references must be resolved at link time (using the above search order). Even though the linker produces an executable file for and unsuccessful link session, an error bit is set in the header which prevents the loader from running the executable.

90H or 91H PUBDEF Public Names Definition Record

Description:

The PUBDEF record contains a list of public names. It makes items defined in this object module available to satisfy external references in other modules with which it is bound or linked.

The symbols are also available for export if so indicated in an EXPDEF comment record.

Record format:

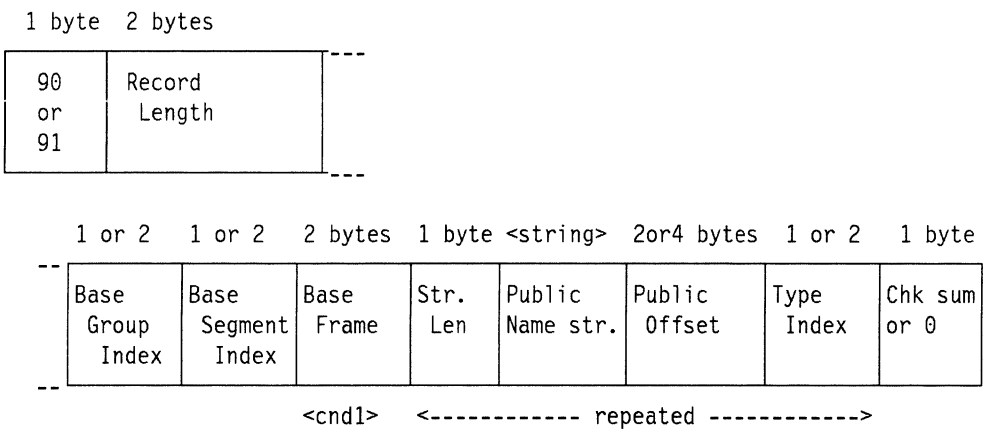


Figure 19. PUBDEF Public Names Definition Record

Base Group, Base Segment and Base Frame

The base group and segment are indices specifying previously defined SEGDEF and GRPDEF records. The group index may be zero, meaning that no group is associated with this PUBDEF record.

The Base Frame field is present only if the Base Segment is zero, but the content of the Base Frame is always ignored by the linker.

The Segment Index is normally nonzero and no Base Frame is present.

The Base Frame is normally used for absolute addressing when the Group and Segment Index are both zero. Absolute addressing is not fully supported in the linker.

Public name, Public Offset and Type Index

The public name string is in <count, char> form and cannot be null. The maximum length of a public name is 255 bytes.

The public offset is a 2 or 4 byte numeric field containing the offset of the location referred to by the public name. This offset is assumed to lie within the segment, group or frame specified in the public base field.

The Type Index field is encoded in index format; it contains either debug type information or zero. This field is ignored by the OS/2 2.0 linker.

NOTES:

All defined functions and initialized global variables generate PUBDEF records.

Any PUBDEF records in an object module must appear after the GRPDEF and SEGDEF records to which they refer.

The IBM C Compiler will generate PUBDEF records for all defined functions and initialized global variables. Globals for scalars that are initialized to zero produce COMDEF records.

Record type 90H uses 16-bit encoding of the Public Offset, but it is zero-extended to 32 bits if applied to Use32 segments.

94H or 95H LINNUM Line Number Record

Description:

The LINNUM record relates line number within language source statements to addresses in the object code.

Record format:

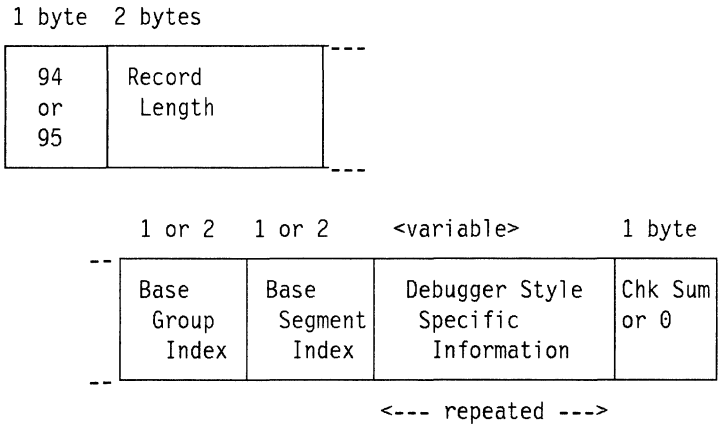


Figure 20. LINNUM line number record

Associates a source line number with translated code or data. The LINNUM record is only generated when the debug option is selected and is therefore specific to the debug information. Refer to the specific debug documentation for more information.

Base Group and Base Segment

The Base group and Base segment are indices specifying previously defined GRPDEF and SEGDEF records. The group index is ignored. The segment index must be nonzero unless the debugger style is version 3 or greater of the IBM PM debugger format.

96H LNAMES List of Names Record

Description:

The LNAMES record is a list of names that can be referenced by subsequent SEGDEF and GRPDEF records in the object module.

The names are ordered by occurrence and referenced by index from subsequent records. More than one LNAMES record may appear. The names themselves are used as segment, class and group names.

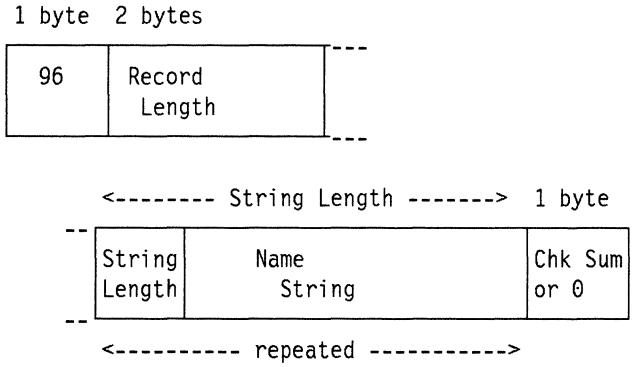


Figure 21. LNAMES list of names record

Each name appears in count/char format, and a null name is valid. The character set is ASCII.

NOTES:

The linker imposes a limit of 255 logical names per object module.

Any LNAMES records in an object module must appear before the records that refer to them. Because it does not refer to any other type of object record, an LNAMES record usually appears near the start of an object module.

98H or 99H SEGDEF Segment Definition Record

Description:

The SEGDEF record describes a logical segment in an object module. It defines the segment's name, length and alignment, as well as the way the segment can be combined with other logical segments at bind, link and load time.

Object records that follow the SEGDEF record can refer to it to identify a particular segment. The SEGDEF records are ordered by occurrence and are referenced by segment indexes (origin 1) in subsequent records.

Record format:

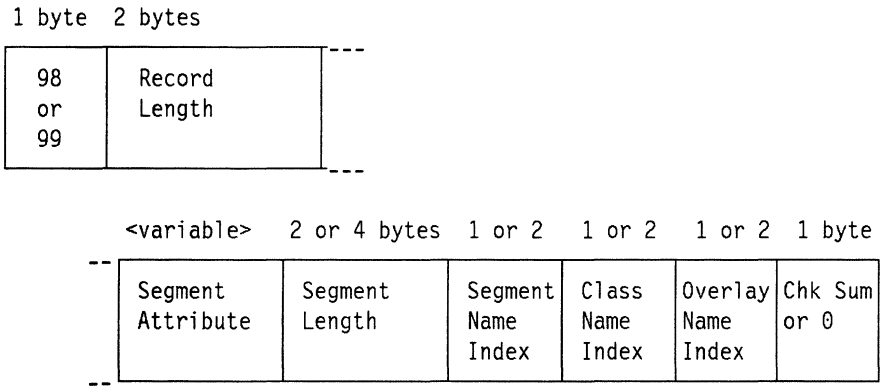
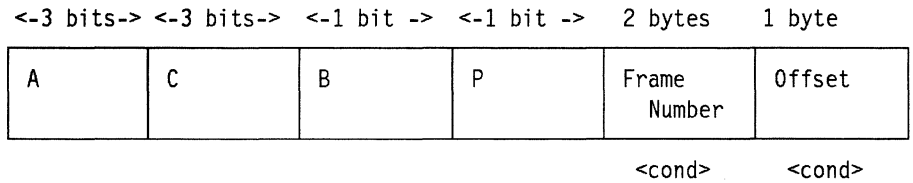


Figure 22. SEGDEF segment definition record

Segment Attributes

The segment attribute is bit-significant; the layout is:



The fields have the following meaning:

A alignment, a 3-bit field, which specifies the alignment required when this program segment is placed within a logical segment. Values are:

- **0** absolute segment
- **1** relocatable, byte aligned
- **2** relocatable, word (2 byte, 16-bit) aligned
- **3** relocatable, paragraph (16 byte) aligned
- **4** relocatable, aligned on page (4K byte) boundary.
- **5** relocatable, aligned on double word (4 byte) boundary
- **6** not supported
- **7** not defined

The new values are A = 4 and A = 5. Dword alignment is expected to be useful as 32-bit memory paths become more prevalent. Page-align maps to the 80386 hardware page size of 4K bytes.

C combination, a 3-bit field, which determines the way the program segment is mapped into a logical segment. Values are:

0 private, do not combine with any other program segment
1 reserved
2 public, combine by appending at an offset which meets the alignment requirement
3 reserved
4 same as C = 2 (public)
5 stack, combine as for C = 2.
6 common, combine by overlay using maximum size
7 same as C = 2 (public)

B big, used as the high order bit of the segment length field. If this bit is set the segment length value must be zero and the segment's size is 2ê32 or 4Gbytes long.

P Holds the descriptor table B/D bit value (this is the descriptor table D bit for code segments and the B bit for data segments).

If zero, then segment is no larger than 64K (if data) and 16-bit addressing and operands are the default (if code). This is a Use16 segment.

If not zero, then the segment is no larger than 64k (if data) and 32-bit addressing and operands are the default (if code). This is a Use32 segment.

Note that this is the only method for defining Use32 segments.

Segment Length

The Segment Length is a 2 or 4 byte numeric quantity and specifies the number of bytes in this program segment.

NOTE For record type 98H, the length can be from 0 to 64K; if a segment is exactly 64KB in size, segemtn length should be 0 and the B field in the ACPB byte should be 1. For record type 99H, the length can be from 0 to 4G; If segment is exactly 4Gbytes in size, segment length should be set to zero and the B field in the ACBP byte should be set to 1.

Segment Name Index, Class Name Index, Overlay Name Index

The three name indices refer to names that appeared in previous LNAMES record(s). The linker ignores the overlay name index. The full name of a segment consists of the segment and class names. Segments in different object modules are normally combined according to the A and C values if their full names are identical. These indices must be nonzero, although the name itself may be null.

The segment name index identifies the segment with a name. the name need not be unique -- other segments of the same name will be concatenated onto the first segment with that name. The name may have been assigned by the programmer, or it may have been generated by the compiler.

The class name index identifies the segment with a class name (such as CODE, DATA or STACK). The linker places segments with the same class name into a contiguous area of memory in the run-time memory map.

The overlay index is ignored by the linker.

Notes:

The linker imposes a limit of 255 SEGDEF records per object module.

The following name/class combinations are reserved:

$$TYPE Reserved for Debug types.
$$SYMBOLS Reserved for Debug names.
CODE32 Reserved for IBM C Compiler.
DATA32 Reserved for IBM C Compiler.
CONST32 Reserved for IBM C Compiler.
BSS32 Reserved for IBM C Compiler.
DGROUP32 Reserved for IBM C Compiler.

9AH GRPDEF Group Definition Record

Description:

The GRPDEF record causes the program segments defined by SEGDEF records to be collected together (grouped). For OS/2 2.0, the segments are combined into a logical segment which is to be addressed through a single selector for flat memory.

Record format:

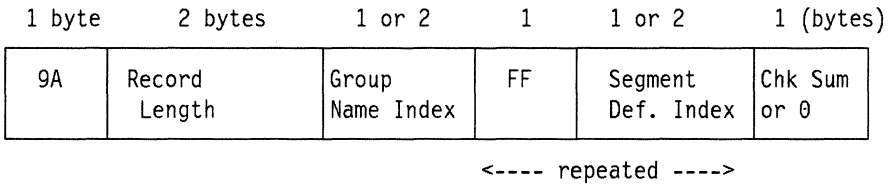


Figure 23. GRPDEF group definition record

This record causes the program segments identified by SEGDEF records to be collected together (grouped) within a logical segment which is to be addressed through a single selector.

Group Name

The group name is specified as an index into a previously defined LNAMES name and must be nonzero.

Groups from different object modules are coalesced if their names are identical.

Group Components

The group's components are segments, specified as indices into previously defined SEGDEF records.

The first byte of each group component is a type field for the remainder of the component. The linker requires a type value of FFH and always assumes that the component contains a segment index value.

The component fields are usually repeated so that all segments constituting a group can be included in one GRPDEF record.

Notes:

This record is frequently followed by a THREAD fixup.

The linker imposes a limit of 31 GRPDEF records in a single object module and limits the total number of group definitions across all object modules to 31.

An example of a group for the IBM C Compiler is DGROUP32 which groups DATA32, CONST32 and BSS32.

The linker does special handling of the pseudo-group ***FLAT***. All address references to this group are made as offsets from the virtual zero address, which is the start of the memory image of the executable.

9CH or 9DH FIXUPP Fixup Record

Description:

The FIXUPP record contains information that allows the linker to resolve (fix up) and eventually relocate references between object modules. FIXUPP records describe the LOCATION of each address value to be fixed up, the TARGET address to which the fixup refers and the FRAME relative to which the address computation is performed.

Record format:

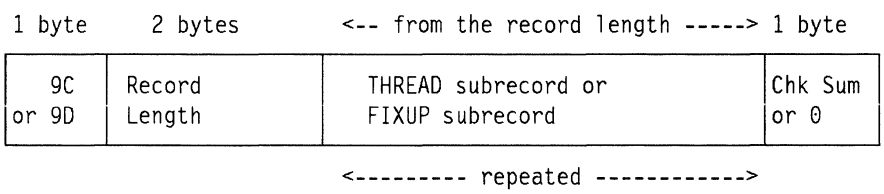


Figure 24. FIXUPP Fixup Record

Record type 9DH is new for LINK386; it has a Target Displacement field of 32 bits rather than 16 bits, and the LOC field of the LOCAT word has been extended to $ bits (using the previously unused higher-order 'S' bit) to allow new LOC values of 9, 11 and 13.

Each subrecord in a FIXUPP object record either defines a thread for subsequent use, or refers to a data location in the nearest previous LEDATA or LIDATA record. The high order bit of the subrecord determines the subrecord type: if the high order bit is 0, the subrecord is a THREAD subrecord; if the high order bit is 1, the subrecord is a FIXUP subrecord. Subrecords of different types may be mixed within one object record.

Information that determines how to resolve a reference can be specified explicitly in a FIXUP subrecord, or can be specified within a FIXUP subrecord by a reference to a previous THREAD subrecord. A THREAD subrecord describes only the method to be used by the linker to refer to a particular target or frame. Because the same THREAD can be referenced in several subsequent FIXUP subrecords, a FIXUPP object record that uses THREADs may be smaller than one in which THREADs are not used.

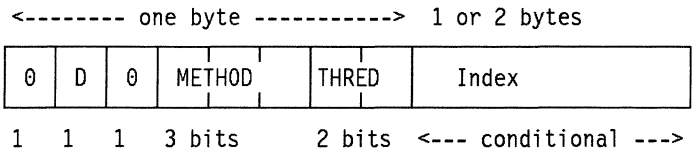
THREAD subrecords can be referenced in the same object record in which they appear and also in subsequent FIXUPP object records.

THREAD

There are 4 frame threads and 4 target threads; not all need be defined and they can be redefined by later THREAD subrecords in the same or later FIXUPP object records. The frame threads are used to specify the Frame Datum field in a later FIXUP subrecord: the target threads are used to specify the Target Datum field in a later FIXUP subrecord.

A THREAD subrecord defines a thread, and does not require that a previous LEDATA or LIDATA record occur.

The layout of the THREAD subrecord is as follows:



where:

0 The high order bit is zero to indicate that this is a THREAD subrecord.

D is 0 for a target thread, 1 for a frame thread

METHOD is a 3-bit field.

For target threads, only the lower two bits of the field are used; the high-order bit of the method is derived from the P bit in the FixDat field of the FIXUP subrecords that refer to this thread. The full list of methods is given here for completeness. This field determines the kind of index required to specify the Target Datum.

T0 specified by a SEGDEF index
T1 specified by a GRPDEF index
T2 specified by a EXTDEF index
T3 specified by an explicit frame number (not supported by the linker)
T4 specified by a SEGDEF index only; the displacement in the FIXUP subrecord is assumed to be 0.
T5 specified by a GRPDEF index only; the displacement in the FIXUP subrecord is assumed to be 0.
T6 specified by a EXTDEF index only; the displacement in the FIXUP subrecord is assumed to be 0.

The index type specified by the target thread method is encoded in the index field.

For frame threads, the method field determines the Frame Datum field of subsequent FIXUP subrecords that refer to this thread. Values for the method field are:

F0 the FRAME is specified by a SEGDEF index
F1 the FRAME is specified by a GRPDEF index
F2 the FRAME is specified by a EXTDEF index. The linker determines the FRAME from the external name's corresponding PUBDEF record in another object module, which specifies either a logical segment or a group.
F3 invalid (The FRAME is identified by an explicit frame number; this is not supported by the linker)
F4 the FRAME is determined by the segment index of the previous LEDATA or LIDATA record (ie the segment in which the location is defined).
F5 the FRAME is determined by the TARGET's segment, group or external index
F6 invalid

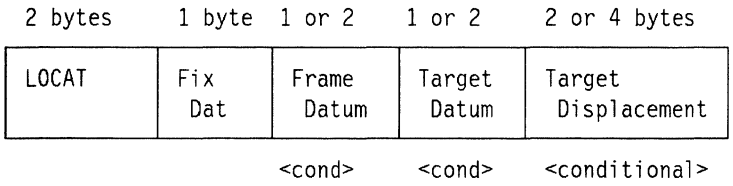
The index field is present for frame methods F0, F1, and F2 only.

THRED is a 2-bit field and determines the thread number (0 through 3, for the 4 threads of each kind).

Index is a conditional field that contains an index value that refers to a previous SEGDEF, GRPDEF or EXTDEF record. The field is only present if the thread method is 0, 1 or 2. If method 3 were supported by the linker, the Index field would contain an explicit frame number.

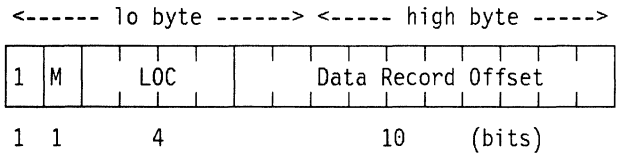
FIXUP

A FIXUP subrecord gives the how/what/why/where/who information required to convert a reference when program segments are combined or placed within logical segments. It applies to the nearest previous LEDATA or LIDATA record, which must be defined before the FIXUP. This FIXUP subrecord is as follows:



where the LOCAT field has an unusual format. Contrary to the usual byte order in Intel data structures, the most significant bits of the LOCAT field are found in the low-order, rather than the high-order byte.

The LOCAT field is:



where:

1 the high bit of the low-order byte is set to indicate a FIXUP subrecord.

M is the mode, M = 1 for segment-relative and M = 0 for self-relative fixups

LOC is a 4-bit field which determines what type of location is to be fixed up:

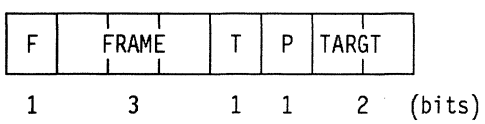
- **0** Low-order byte (8-bit displacement or low byte of 16-bit offset)
- **1** 16-bit Offset
- **2** 16-bit Base - logical segment base (selector)
- **3** 32-bit Long pointer (16-bit base : 16-bit offset)
- **4** Hi-order byte (high byte of 16-bit offset) No linker support for this type.
- **5** 16-bit loader-resolved offset, treated as LOC = 1 by the linker

  CONFLICT PharLap OMF uses LOC = 5 to indicate a 32-bit offset, where Microsoft and IBM use LOC = 9.
- **6** not defined, reserved

  CONFLICT PharLap OMF uses LOC = 6 to indicate a 48-bit pointer (16-bit base : 32-bit offset) where Microsoft and IBM use LOC = 11.
- **7** not defined, reserved
- **9** 32-bit offset
- **11** 48-bit pointer (16-bit base : 32-bit offset)
- **13** 32-bit loader-resolved offset, treated as LOC = 9 by the linker

Data Record Offset The Data record offset indicates the position of the location to be fixed up in the LEDATA or LIDATA record immediatly preceding the FIXUPP record. This offset indicates either a byte in the data field of an LEDATA record or a data byte in the content field of an iterated data block in an LIDATA record.

The FixDat bit layout is:



and is interpreted as follows:

F If F = 1, the frame is given by a frame thread whose number is in the FRAME field (modulo 4). There is no frame datum field in the subrecord.

If F = 0, the frame method (in the range F0 to F5) is explicitly defined in this FIXUP subrecord. The method is stored in the FRAME field.

FRAME

3-bit numeric, interpreted according to the F bit. The Frame Datum field is present and is an index field for frame methods F0, F1, and F2 only.

T If T = 1 the target is defined by a target thread whose thread number is given in the 2-bit TARGT field. The TARGT field contains a number between 0 and 3 that refers to a previous thread field containing the target method. The P bit, combined with the two low-order bits of the method field in the THREAD subrecord, determines the target method.

If T = 0 the target is specified explicitly in this FIXUP subrecord. In this case, the P bit and the TARGT field can be considered a 3-bit field analogous to the FRAME field.

P Determines whether the target displacement field is present.

If P = 1 there is no displacement field.

If P = 0, the displacement field is present.

TARGT is a 2-bit numeric, which gives the lower two bits of the target method (if T = 0) or gives the target thread number (if T = 1).

Frame Datum is an index field that refers to a previous SEGDEF, GRPDEF or EXTDEF record, depending on the FRAME method.

Target Datum contains a segment index, a group index or an external index depending on the TARGET method.

Target Displacement a 16-bit or 32-bit field is present only if the P bit in the FixDat field is set to to 0, in which case the Target Displacement field contains the offset used in methods 0, 1 and 2 of specifying a TARGET.

Notes:

FIXUPP records are used to fix references in the immediately preceding LEDATA, LIDATA or COMDAT record.

The FRAME is the translator's way of telling the linker the contents of the segment register used for the reference; the TARGET is the item being referenced whose address was not completely resolved by the translator. In protect mode, the only legal segment register value are selectors; every segment and group of segments is mapped through some selector and addressed by offset within the underlying memory defined by that selector.

A0H or A1H LEDATA Logical Enumerated Data Record

Description:

This record provides contiguous binary data -- executable code or program data -- which is part of a program segment. The data is eventually copied into the program's executable binary image by the linker.

The data bytes may be subject to relocation or fix-up as determined by the presence of a subsequent FIXUPP record but otherwise requires no expansion when mapped to memory at run time.

Record Format:

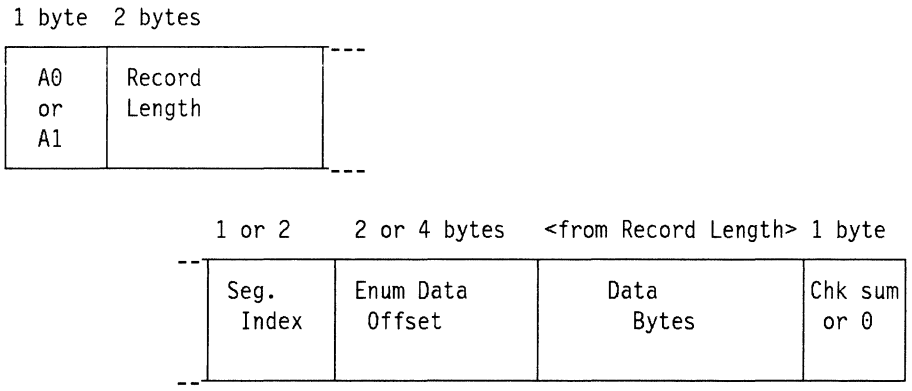


Figure 25. LEDATA logical enumerated data record

Record type A1H is new for LINK386; it has an Enumerated Data Offset field of 32 bits rather than 16 bits.

Segment Index

The SegIndex must be nonzero and is the index of a previously defined SEGDEF record. This is the segment into which the data in this LEDATA record is to be placed.

Enumerated Data Offset

The enumerated data offset is a 2 or 4 byte field (depending on the record type) which determines the offset at which the first data byte is to be placed relative to the start of the SEGDEF segment. Successive data bytes occupy successively higher locations.

Data Bytes

The maximum number of data bytes is 1024, so that a FIXUPP location field, which is 10 bits, can reference any of these data bytes. The number of data bytes is computed as the RecLength minus 5 minus the size of the SegIndex field (1 or 2 bytes).

NOTES:

Record type A1H has offset stored as a 32-bit numeric. Record type A0 encodes the offset value as a 16-bit numeric (zero extended if applied to a Use32 segment).

If the LEDATA requires fixup, a FIXUPP record must immediately follow the LEDATA record.

A2H or A3H LIDATA Logical Iterated Data Record

Description:

Like the LEDATA record, the LIDATA record contains binary data -- executable code or program data. The data in an LIDATA record, however, is specified as a repeating pattern (iterated), rather than by explicit enumeration.

The data in an LIDATA record may be modified by the linker if the LIDATA record is immediately followed by a FIXUPP record.

Record format:

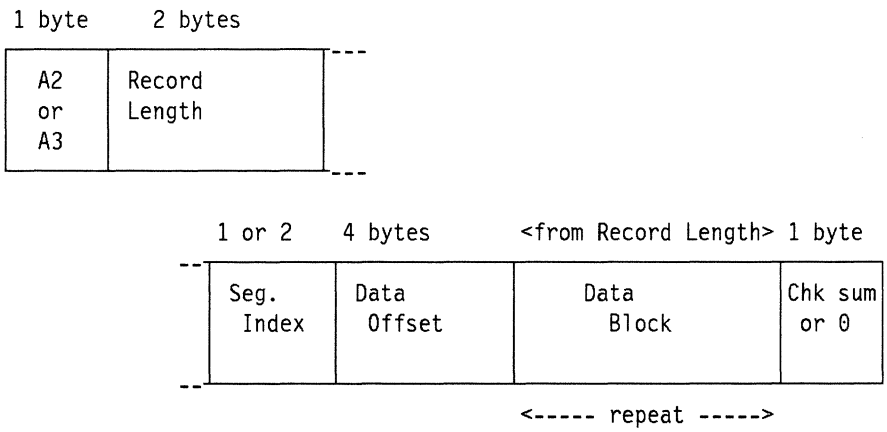


Figure 26. LIDATA Logical Iterated Data Record

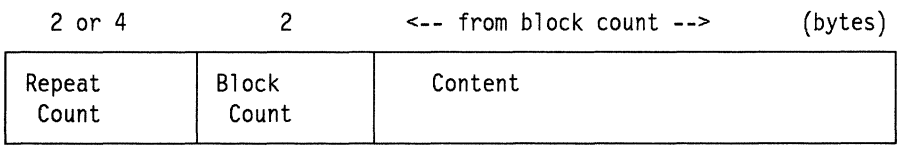
Record type A3H is new for LINK386; it has Iterated Data Offset and Repeat Count fields of 32 bits rather than 16 bits.

Segment Index and Data Offset

The segment index and data offset (2 or 4 bytes) are the same as for an LEDATA record. The index must be nonzero.

Data Block

The data blocks have the following form:



Repeat Count

The Repeat Count is a 16-bit or 32-bit value which determines the number of repeats of the content field. The Repeat Count is 32 bits only if the record type is A3.

CONFLICT:The PharLap OMF uses a 16-bit repeat count even in 32-bit records.

Block Count

The Block Count is a 16-bit word whose value determines the interpretation of the content portion, as follows:

0 indicates that the content field that follows is a one byte "count" value followed by "count" data bytes. The data bytes will be mapped to memory, repeated Repeat Count times.

!=0 indicates the content field that follows is comprised of one or more Data Blocks. The Block Count value specifies the number of Data Blocks (recursive definition).

Notes:

A subsequent FIXUPP record may occur; the fixup is applied before the iterated data block is expanded. It is a translator error for a fixup to reference any of the count fields.

B0H COMDEF Communal Names Definition Record

Description:

The COMDEF record declares a list of one or more communal variables (uninitialized static data, or data that may match initialized static data in another compilation group).

The size of such a variable the the maximum size defined in any module naming the variable as communal or public. The placement of communal variables is determined by the data type using established conventions (see data type and communal length below).

Record format:

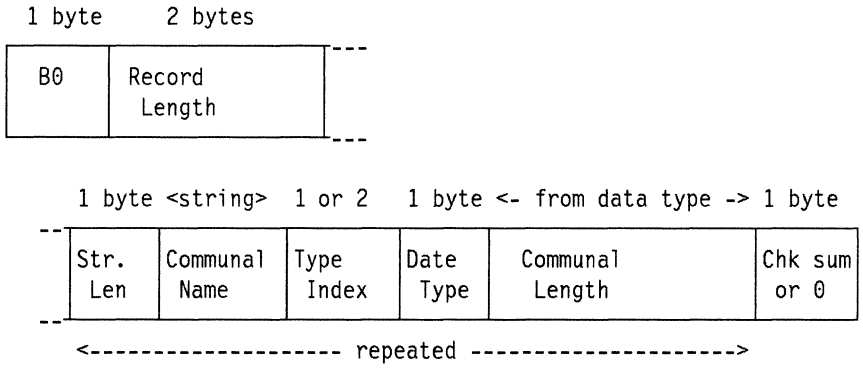


Figure 27. COMDEF Communal Names Definition Record

Communal name

The name is in <count, char> string format (and name may be null). Near and Far communals from different object files are matched at bind or link time if their names agree; the variable's size is the maximum of the sizes specified (subject to some constraints, as documented below).

Type Index

Encodes symbol information; it is parsed as an index field (one or 2 bytes), and not inspected by the linker.

Data Type and Communal Length

The data type field indicates the contents of the Communal Length field. All Data type values for Near data indicate that the Communal Length field has only one numeric value: the amount of memory to be allocated for the communal variable. All Data Type values for Far data indicate that the Communal Length field has two numeric values: the first is the number of elements and the second is the element size.

The DataType is one of the following hex values:

61H FAR data; length specified as number of elements followed by element size in bytes.
62H NEAR data; length specified as number of bytes.

The communal length is a single numeric or a pair of numeric fields (as specified by the Data Type), encoded as follows:

1 byte value 0 through 128 (80 hex)
3 byte byte 81 hex, followed by a 16-bit word whose value is used (range 0 to 64K-1)
4 byte byte 84 hex, followed by a 3 byte value (range 0 to 16M-1)
5 byte byte 88 hex, followed by a 4 byte value (range -2G-1 to 2G-1, signed)

Groups of name, type index, segment type and communal length fields can be repeated so that more than one communal variable can be declared in the same COMDEF record.

Notes:

If a public or exported symbol with the same name is found in another module with which this is bound or linked, the linker gives a multiple defined symbol error message,

Communal variables cannot be resolved to dynamic links (ie, imported symbols).

The records are ordered by occurrence, together with the items named in EXTDEF records (for reference in FIXUPS).

The IBM C Compiler generates COMDEF's for all uninitialized global data and for global scalars initialized to zero.

B2H or B3H BAKPAT Backpatch Record

Description:

This record is for backpatches to locations which cannot be conveniently handled by a FIXUPP at reference time. For example, forward references in a one-pass compiler. It is essentially a specialized fixup.

Record format:

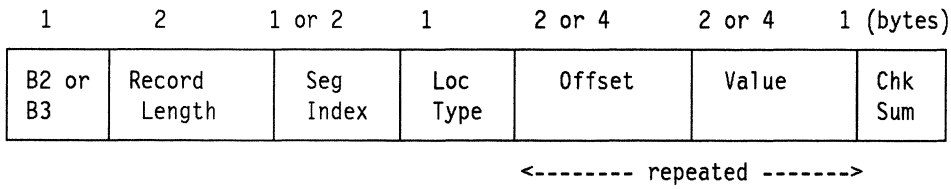


Figure 28. BAKPAT Backpatch record

SegIndex

Segment index to which all "backpatch" fixupps are to be applied. Note that, in contrast to FIXUPs, these records need not follow the data record to be fixed up. Hence, the segment to which the backpatch apllies must be specified explicitly.

LocTyp

Type of location to be patched; the only valid values are:

0	8-bit lobyte
1	16-bit offset
9	32-bit offset, record type B3 only

Offset and value

These fields are 32-bits for record type B3, 16-bit for B2.

The Offset specifies the loaction to be patched (as an offset into the segdef whose index is SegIndex).

The associated Value is added to the location being patched (unsigned addition, ignoring overflow). The Value field is fixed length (16-bit or 32-bit, depending on the record type) to make object module processing easier.

Notes:

BAKPAT records can occur anywhere in the object module following the SEGDEF record to which they refer. They do not have to immediately follow the appropriate LEDATA record as FIXUPP record do.

These records are buffered by the linker in Pass 2 until the end of the module, after applying all other FIXUPPs. The linker then processes the records as fixups.

B4H or B5H LEXTDEF Local External Names Definition Record

Description:

This record is identical in form to the EXTDEF record described earlier. However, the symbols named in this record are not visible outside the module in which they are defined.

Record format:

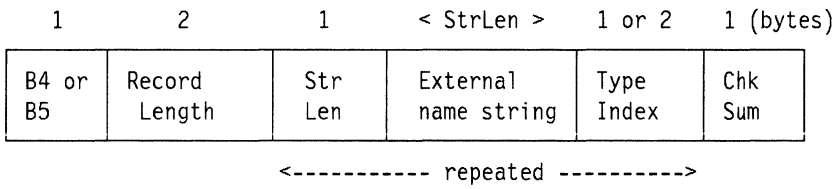


Figure 29. LEXTDEF Local External Names Definition Record

Notes:

There is no semantic difference between the B4 and B5 flavors.

These records are associated with LPUBDEF and LCOMDEF records, ordered with the EXTDEF records by occurence, so that they may be referenced by external index for fixups.

The name string, when stored in LINK's internal data structures, is encoded with spaces and digits at the beginning of the name.

B6H or B7H LPUBDEF Local Public Names Definition Record

Description:

This record is identical in form to the PUBDEF record described earlier. However, the symbols named in this record are not visible outside the module in which they are defined.

Record format:

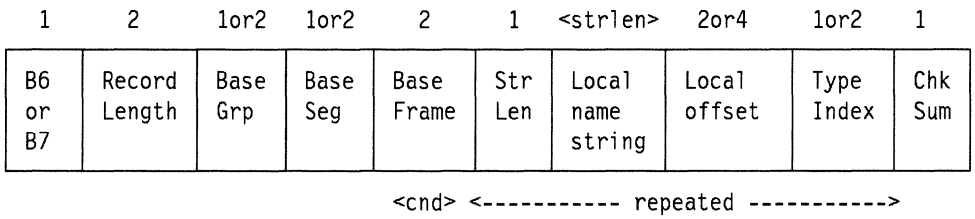


Figure 30. LPUBDEF Local Public Names Definition Record

B8H LCOMDEF Local Communal Names Definition Record

Description:

This record is identical in form to the COMDEF record described earlier. However, the symbols named in this record are not visible outside the module in which they are defined.

Record format:

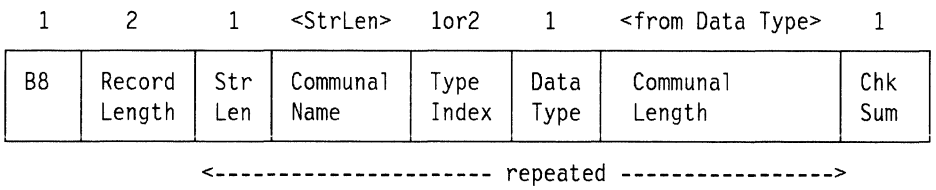


Figure 31. LCOMDEF Local Communal Names Definition Record

C2H or C3H COMDAT Initialized Communal Data Record

Description:

The purpose of the COMDAT record is to combine logical blocks of code and data which may be duplicated across a number of compiled modules.

Record format:

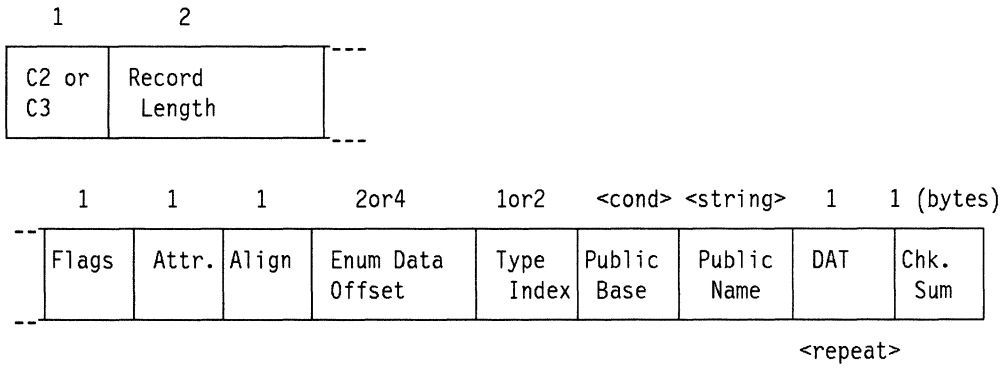


Figure 32. COMDAT initialized communal data record

Flags

This field contains three defined bits:

01H - Continuation bit. If clear, then this COMDAT record establishes a new instance of the COMDAT variable, otherwise the data is a continuation of the previous COMDAT of the symbol.

02H - Iterated data bit. If clear, the Dat field contains enumerated data, otherwise the Dat field contains iterated data, as in an LIDATA record.

04H - Local COMDAT. The public name is local.

Attr

This field contains two 4-bit fields: the selection criteria to be used, the allocation type and the ordinal specifying the type of allocation to be performed. Values are:

Selection criteria (high-order 4 bits):

00H - No match - only one instance of this comdat allowed.

10H - Pick any - pick any instance of this COMDAT record.

20H - Same size - pick any, but instances must have the same length or linker will generate an error.

30H - Exact Match - pick any, but checksums of instances must match of linker will generate an error. Fixups are ignored.

40H - F0H - reserved.

Allocation Type (low-order 4 bits):

00H - Explicit - allocate in the segment specified in the ensuing public base field.

01H - Far Code - allocate as CODE16. The linker will create segments to contain all COMDAT's of this type.

02H - Far DATA - allocate as DATA16. The linker will create segments to contain all COMDAT's of this type.

03H - CODE32 - allocate as CODE32. The linker will create segments to contain all COMDAT's of this type.

04H - DATA32 - allocate as DATA32. The linker will create segments to contain all COMDAT's of this type.

05H - 0FH - Reserved.

Align

These codes are based on the ones used by the SEGDEF record:

0 - use value from SEGDEF.

1 - byte aligned.

2 - word (2 byte) aligned.

3 - paragraph (16 byte) aligned.

4 - 4K page aligned.

5 - dword (4 byte) aligned.

6 - not defined.

7 - not defined.

Enum Data Offset

This field specifies an offset relative to the beginning location of the symbol specified in the public name field and defines the relative location of the first byte of the DAT field. Successive data bytes in the DAT field occupy higher locations of memory. This works very much like the offset field in an LEDATA record, but instead of an offset relative to a segment, this is relative to the beginning of the COMDAT symbol.

Type Index

The type index field is encoded in index format; it contains either debug type information or an old-style TYPDEF index. If this index is zero, there is no associated type data. Old-style TYPDEF indices are ignored by the linker. Present linkers do no type checking.

Public Base

This field is conditional and is identical to the public base stored in the public base field in the PUBDEF record. This field is only present if the allocation type field specifies explicit allocation.

Public Name

This field is a regular length prefixed name.

Dat

The Dat field provides up to 1024 consecutive bytes of data. If there are fixups, they must be emitted in a FIXUPP record that follows the COMDAT record. The data can be either enumerated or iterated, depending on the flags field.

Notes:

While creating addressing frames, the linker will add the COMDAT data to the appropriate logical segments, adjusting their sizes. At that time the offset at which the data will go inside the logical segment will be calculated. Next, the linker will create physical segments from adjusted logical segments reporting any 64K boundary overflows.

If the allocation type is not explicit, COMDAT code and data is accumulated by the linker and broken up into segments, so that the total may exceed 64K.

In pass two, only the selected occurrence of COMDAT data will be stored in the VM, fixed up and later written into the .EXE file.

C4H or C5H LINSYM Symbol Line Numbers Record

Description:

This record will be used to output numbers for functions specified via COMDATs.

Record format:

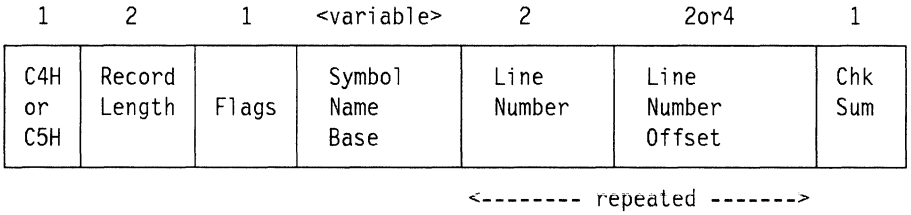


Figure 33. COMDAT initialized communal data record

Flags

This field contains three defined bits:

01H Continuation bit. If clear, then this COMDAT record establishes a new instance of the COMDAT variable, otherwise the data is a continuation of the previous COMDAT of the symbol.

04H Local COMDAT

The **Symbol Name Base** is a length-preceded name of the base of the LINSYM record.

The **Line Number** is an unsigned number in the range 0 to 65535.

The **Line Number Offset** field is the offset relative to tha base specified by the symbol name base. The size of this field depends on the record type.

Notes:

Record type C5H identical to C4H except that the Line Number Offset field is 4 bytes instead of 2.

Thie record is used to output line numbers for functions specified via COMDATs. Often, the residing segment as well as the relative offsets of such function is unknown at compile time, in that the linker is the final arbitrator of such information. For such cases the compiler will generate this record to specify the line number/offset pairs relative to a symbolic name.

Thie record will also be used to discard duplicate linnum information. If the linker encounters two LINSYM records with matching symbolic names, the linker will keep the first set of linnums and discard all subsequent LINSYM records of that name.

C6H ALIAS Alias Definition Record

Description:

This record has been introduced to support link-time aliasing, or a method by which compilers or assembles may direct the linker to substitute all references to one symbol for another.

Record format:

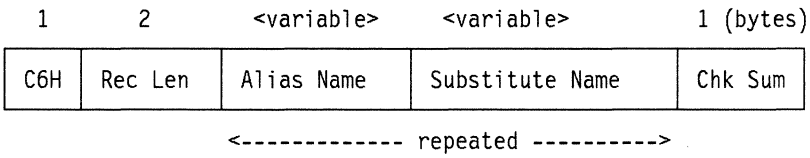


Figure 34. ALIAS Alias Definition Record

The **Alias Name** field is a regular length-preceded name of the alias symbol.

The **Substitute Name** field is a regular length-preceded name of the substitute symbol.

Notes:

The record will consist of two symbolic names: the alias symbol and the substitute symbol. The alias symbol behaves very much like a PUBDEF in that it must be unique. If a PUBDEF of an alias symbol is encountered or another ALIAS record with a different substitute symbol is encountered, a redefinition error should be emitted by the linker.

When attempting to satisfy an external reference, if an ALIAS record whose alias symbol matches is found, the linker will halt the search for alias symbol definitions and will attempt to satisfy the reference with the substitute symbol.

All ALIAS records must appear before the link pass 2 record.

C8H or C9H NBKPAT Named BackPatch Record

Description:

The Named Backpatch record is like a BAKPAT record, except that it refers to a COMDAT, by name, rather than an LIDATA or LEDATA record.

Record format:

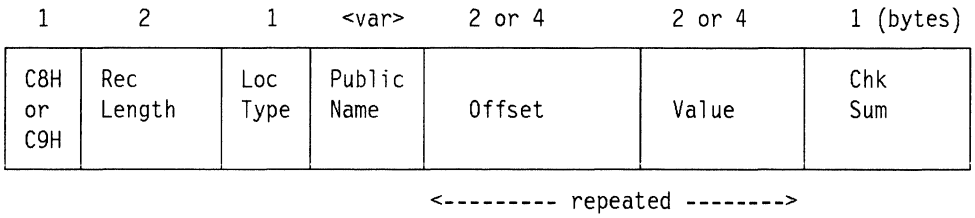


Figure 35. NBKPAT Named Backpatch Record

LocType

Type of location to be patched; the only valid values are:

0	8-bit byte
1	16-bit word
2	32-bit dword, record type C9 only
0x80	local COMDAT

Public Name

Length-preceded name of the COMDAT to back patch.

Offset and Value

These fields are 16-bits for record type C8, 32-bits for C9.

The Offset specifies the location to be patched, as an offset into the COMDAT.

The associated Value is added to the location being patched (unsigned addition, ignoring overflow). The Value field is fixed length (16-bit or 32-bit, depending on the record type) to make object module processing easier.

LX - Linear eXecutable Module Format Description

Revision codes:

revision 1 - Library termination.

revision 2 - Sector Align and Exepack support.

revision 3 - Address Based linking.

revision 4 - OS/2 2.0 PM Debugger (IBM) support.

Revision 8 = Added ITERDATA2 definition and minor corrections

32-bit Linear EXE Header

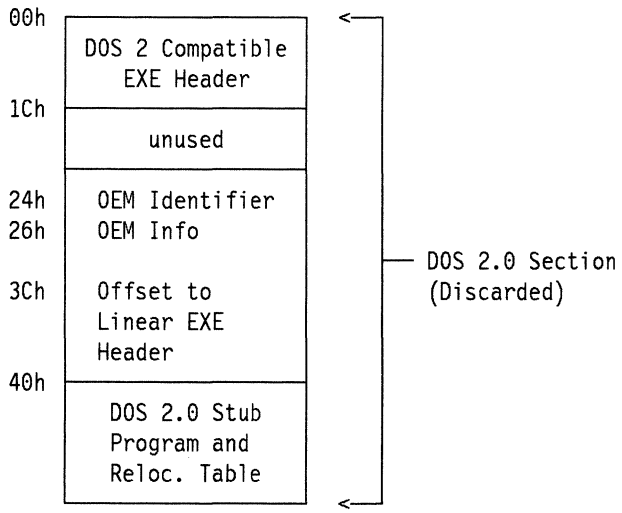


Figure 36. Dos 2.0 Section (Discarded)

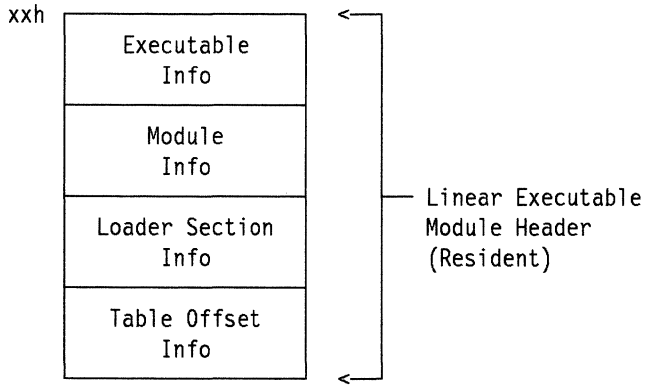


Figure 37. Linear Executable Module Header (Resident)

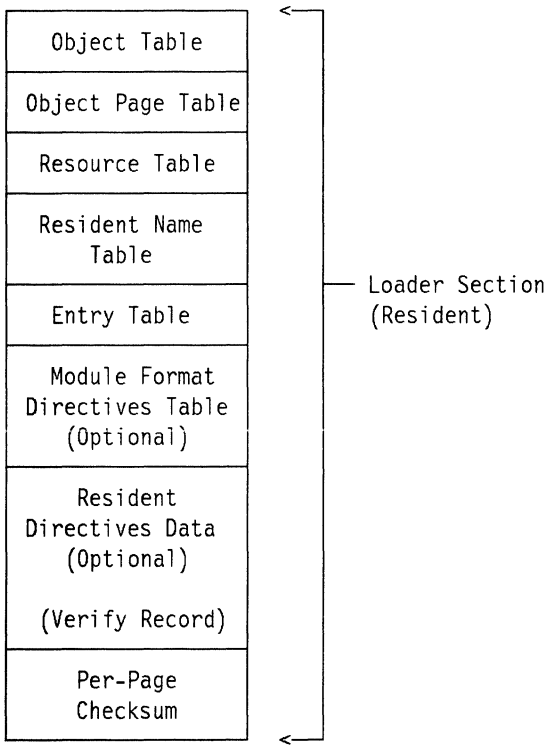


Figure 38. Loader Section (Resident)

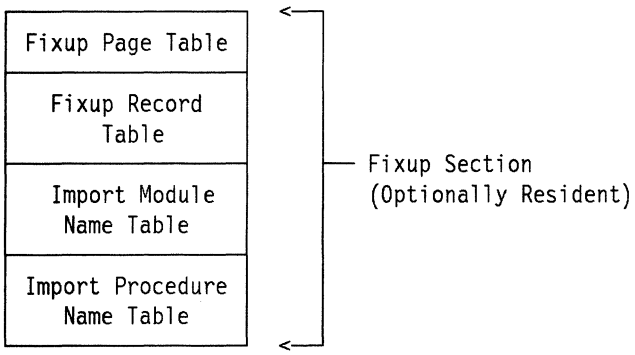


Figure 39. Loader Section (Resident)

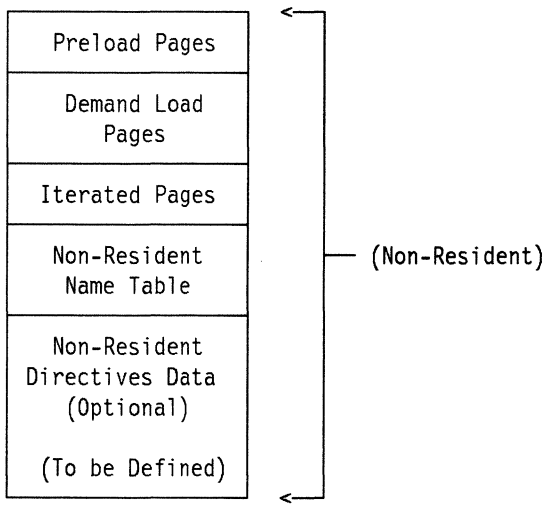


Figure 40. Non-Resident Section

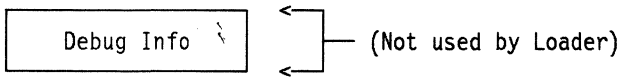


Figure 41. Not used by the Loader

LX Header

00h	"L" "X"	B-ORD	W-ORD	FORMAT LEVEL
08h	CPU TYPE	OS TYPE		MODULE VERSION
10h	MODULE FLAGS			MODULE # OF PAGES
18h	EIP OBJECT #			EIP
20h	ESP OBJECT #			ESP
28h	PAGE SIZE			PAGE OFFSET SHIFT
30h	FIXUP SECTION SIZE			FIXUP SECTION CHECKSUM
38h	LOADER SECTION SIZE			LOADER SECTION CHECKSUM
40h	OBJECT TABLE OFF			# OBJECTS IN MODULE
48h	OBJECT PAGE TABLE OFF			OBJECT ITER PAGES OFF
50h	RESOURCE TABLE OFFSET			#RESOURCE TABLE ENTRIES
58h	RESIDENT NAME TBL OFF			ENTRY TABLE OFFSET
60h	MODULE DIRECTIVES OFF			# MODULE DIRECTIVES
68h	FIXUP PAGE TABLE OFF			FIXUP RECORD TABLE OFF
70h	IMPORT MODULE TBL OFF			# IMPORT MOD ENTRIES
78h	IMPORT PROC TBL OFF			PER-PAGE CHECKSUM OFF
80h	DATA PAGES OFFSET			#PRELOAD PAGES
88h	NON-RES NAME TBL OFF			NON-RES NAME TBL LEN
90h	NON-RES NAME TBL CKSM			AUTO DS OBJECT #
98h	DEBUG INFO OFF			DEBUG INFO LEN
A0h	#INSTANCE PRELOAD			#INSTANCE DEMAND
A8h	HEAPSIZE			STACKSIZE

Figure 42. 32-bit Linear EXE Header

Note: The OBJECT ITER PAGES OFF must either be 0 or set to the same value as DATA PAGES OFFSET in OS/2 2.0. Ie., iterated pages are required to be in the same section of the file as regular pages.

Note: Table offsets in the Linear EXE Header may be set to zero to indicate that the table does not exist in the EXE file and it's size is zero.

"L" "X" = DW Signature word.

The signature word is used by the loader to identify the EXE file as a valid 32-bit Linear Executable Module Format. "L" is low order byte. "X" is high order byte.

B-ORD = DB Byte Ordering.

This byte specifies the byte ordering for the linear EXE format. The values are:

00H - Little Endian Byte Ordering.
01H - Big Endian Byte Ordering.

W-ORD = DB Word Ordering.

This byte specifies the Word ordering for the linear EXE format. The values are:

00H - Little Endian Word Ordering.
01H - Big Endian Word Ordering.

Format Level = DD Linear EXE Format Level.

The Linear EXE Format Level is set to 0 for the initial version of the 32-bit linear EXE format. Each incompatible change to the linear EXE format must increment this value. This allows the system to recognized future EXE file versions so that an appropriate error message may be displayed if an attempt is made to load them.

CPU Type = DW Module CPU Type.

This field specifies the type of CPU required by this module to run. The values are:

01H - 80286 or upwardly compatible CPU is required to execute this module.
02H - 80386 or upwardly compatible CPU is required to execute this module.
03H - 80486 or upwardly compatible CPU is required to execute this module.

OS Type = DW Module OS Type.

This field specifies the type of Operating system required to run this module. The currently defined values are:

00H - Unknown (any "new-format" OS)
01H - OS/2 (default)
02H - Windows[2]
03H - DOS 4.x
04H - Windows 386[2]

MODULE VERSION = DD Version of the linear EXE module.

This is useful for differentiating between revisions of dynamic linked modules. This value is specified at link time by the user.

MODULE FLAGS = DD Flag bits for the module.

The module flag bits have the following definitions.

00000001h = Reserved for system use.
00000002h = Reserved for system use.
00000004h = Per-Process Library Initialization.
The setting of this bit requires the EIP Object # and EIP fields to have valid values. If the EIP Object # and EIP fields are valid and this bit is NOT set, then Global Library Initialization is assumed. Setting this bit for an EXE file is invalid.

[2] Windows is a Registered Trademark of Microsoft Corp.

00000008h = Reserved for system use.
00000010h = Internal fixups for the module have been applied.
The setting of this bit in a Linear Executable Module indicates that each object of the module has a preferred load address specified in the Object Table Reloc Base Addr. If the module's objects can not be loaded at these preferred addresses, then the relocation records that have been retained in the file data will be applied.

00000020h = External fixups for the module have been applied.
00000040h = Reserved for system use.
00000080h = Reserved for system use.
00000100h = Incompatible with PM windowing.
00000200h = Compatible with PM windowing.
00000300h = Uses PM windowing API.
00000400h = Reserved for system use.
00000800h = Reserved for system use.
00001000h = Reserved for system use.
00002000h = Module is not loadable.
When the 'Module is not loadable' flag is set, it indicates that either errors were detected at link time or that the module is being incrementally linked and therefore can't be loaded.

00004000h = Reserved for system use.
00038000h = Module type mask.
00000000h = Program module.
A module can not contain dynamic links to other modules that have the 'program module' type.

00008000h = Library module.
00018000h = Protected Memory Library module.
00020000h = Physical Device Driver module.
00028000h = Virtual Device Driver module.
40000000h = Per-process Library Termination.
The setting of this bit requires the EIP Object # and EIP fields to have valid values. If the EIP Object # and EIP fields are valid and this bit is NOT set, then Global Library Termination is assumed. Setting this bit for an EXE file is invalid.

MODULE # PAGES = DD Physical number of pages in module.

This field specifies the number of pages physically contained in this module. In other words, pages containing either enumerated or iterated data, not invalid or zero-fill pages. These pages are contained in the 'preload pages', 'demand load pages' and 'iterated data pages' sections of the linear EXE module. This is used to determine the size of the other physical page based tables in the linear EXE module.

EIP OBJECT # = DD The Object number to which the Entry Address is relative.

This specifies the object to which the Entry Address is relative. This must be a nonzero value for a program module to be correctly loaded. A zero value for a library module indicates that no library entry routine exists. If this value is zero, then both the Per-process Library Initialization bit and the Per-process Library Termination bit must be clear in the module flags, or else the loader will fail to load the module. Further, if the Per-process Library Termination bit is set, then the object to which this field refers must be a 32-bit object (i.e., the Big/Default bit must be set in the object flags; see below).

EIP = DD Entry Address of module.

The Entry Address is the starting address for program modules and the library initialization and Library termination address for library modules.

ESP OBJECT # = DD The Object number to which the ESP is relative.

This specifies the object to which the starting ESP is relative. This must be a nonzero value for a program module to be correctly loaded. This field is ignored for a library module.

ESP = DD Starting stack address of module.

The ESP defines the starting stack pointer address for program modules. A zero value in this field indicates that the stack pointer is to be initialized to the highest address/offset in the object. This field is ignored for a library module.

PAGE SIZE = DD The size of one page for this system.

This field specifies the page size used by the linear EXE format and the system. For the initial version of this linear EXE format the page size is 4Kbytes. (The 4K page size is specified by a value of 4096 in this field.)

PAGE OFFSET SHIFT = DD The shift left bits for page offsets.

This field gives the number of bit positions to shift left when interpreting the Object Page Table entries' page offset field. This determines the alignment of the page information in the file. For example, a value of 4 in this field would align all pages in the Data Pages and Iterated Pages sections on 16 byte (paragraph) boundaries. A Page Offset Shift of 9 would align all pages on a 512 byte (disk sector) basis. All other offsets are byte aligned.

A page might not start at the next available alignment boundary. Extra padding is acceptable between pages as long as each page starts on an alignment boundary. For example, several alignment boundarys may be skipped in order to start a frequently accessed page on a sector boundary.

FIXUP SECTION SIZE = DD Total size of the fixup information in bytes.

This includes the following 4 tables:

- Fixup Page Table
- Fixup Record Table
- Import Module name Table
- Import Procedure Name Table

FIXUP SECTION CHECKSUM = DD Checksum for fixup information.

This is a cryptographic checksum covering all of the fixup information. The checksum for the fixup information is kept separate because the fixup data is not always loaded into main memory with the 'loader section'. If the checksum feature is not implemented, then the linker will set these fields to zero.

LOADER SECTION SIZE = DD Size of memory resident tables.

This is the total size in bytes of the tables required to be memory resident for the module, while the module is in use. This total size includes all tables from the Object Table down to and including the Per-Page Checksum Table.

LOADER SECTION CHECKSUM = DD Checksum for loader section.

This is a cryptographic checksum covering all of the loader section information. If the checksum feature is not implemented, then the linker will set these fields to zero.

OBJECT TABLE OFF = DD Object Table offset.

This offset is relative to the beginning of the linear EXE header.

OBJECTS IN MODULE = DD Object Table Count.

This defines the number of entries in Object Table.

OBJECT PAGE TABLE OFFSET = DD Object Page Table offset

This offset is relative to the beginning of the linear EXE header.

OBJECT ITER PAGES OFF = DD Object Iterated Pages offset.

This offset is relative to the beginning of the EXE file.

RESOURCE TABLE OFF = DD Resource Table offset.

This offset is relative to the beginning of the linear EXE header.

RESOURCE TABLE ENTRIES = DD Number of entries in Resource Table.

RESIDENT NAME TBL OFF = DD Resident Name Table offset.

This offset is relative to the beginning of the linear EXE header.

ENTRY TBL OFF = DD Entry Table offset.

This offset is relative to the beginning of the linear EXE header.

MODULE DIRECTIVES OFF = DD Module Format Directives Table offset.

This offset is relative to the beginning of the linear EXE header.

MODULE DIRECTIVES = DD Number of Module Format Directives in the Table.

This field specifies the number of entries in the Module Format Directives Table.

FIXUP PAGE TABLE OFF = DD Fixup Page Table offset.

This offset is relative to the beginning of the linear EXE header.

FIXUP RECORD TABLE OFF = DD Fixup Record Table Offset

This offset is relative to the beginning of the linear EXE header.

IMPORT MODULE TBL OFF = DD Import Module Name Table offset.

This offset is relative to the beginning of the linear EXE header.

IMPORT MOD ENTRIES = DD The number of entries in the Import Module Name Table.

IMPORT PROC TBL OFF = DD Import Procedure Name Table offset.

This offset is relative to the beginning of the linear EXE header.

PER-PAGE CHECKSUM OFF = DD Per-Page Checksum Table offset.

This offset is relative to the beginning of the linear EXE header.

DATA PAGES OFFSET = DD Data Pages Offset.

This offset is relative to the beginning of the EXE file.

PRELOAD PAGES = DD Number of Preload pages for this module. Note that OS/2 2.0 does not respect the preload of pages as specified in the executable file for performance reasons.

NON-RES NAME TBL OFF = DD Non-Resident Name Table offset.

This offset is relative to the beginning of the EXE file.

NON-RES NAME TBL LEN = DD Number of bytes in the Non-resident name table.

NON-RES NAME TBL CKSM = DD Non-Resident Name Table Checksum.

This is a cryptographic checksum of the Non-Resident Name Table.

AUTO DS OBJECT # = DD The Auto Data Segment Object number.

This is the object number for the Auto Data Segment used by 16-bit modules. This field is supported for 16-bit compatibility only and is not used by 32-bit modules.

DEBUG INFO OFF = DD Debug Information offset.

This offset is relative to the beginning of the file.

Note: Earlier versions of this doc stated that this offset was from the linear EXE header - this is incorrect.

DEBUG INFO LEN = DD Debug Information length.

The length of the debug information in bytes.

INSTANCE PRELOAD = DD Instance pages in preload section.

The number of instance data pages found in the preload section.

INSTANCE DEMAND = DD Instance pages in demand section.

The number of instance data pages found in the demand section.

HEAPSIZE = DD Heap size added to the Auto DS Object.

The heap size is the number of bytes added to the Auto Data Segment by the loader. This field is supported for 16-bit compatibility only and is not used by 32-bit modules.

STACKSIZE = DD Stack size.

The stack size is the number of bytes specified by:

1. size of a segment with combine type stack
2. STACKSIZE in the .DEF file
3. /STACK link option

The stacksize may be zero.

Note: Stack sizes with byte 2 equal to 02 or 04 (e.g. 00020000h, 11041111h, 0f02ffffh) should be avoided for programs that will run on OS/2 2.0.

Program (EXE) startup registers and Library entry registers

Program startup registers are defined as follows.

EIP = Starting program entry address.

ESP = Top of stack address.

CS = Code selector for base of linear address space.

DS = ES = SS = Data selector for base of linear address space.

FS = Data selector of base of Thread Information Block (TIB).

GS = 0.

EAX = EBX = 0.

ECX = EDX = 0.

ESI = EDI = 0.

EBP = 0.

[ESP+0] = Return address to routine which calls DosExit(1,EAX).

[ESP+4] = Module handle for program module.

[ESP+8] = Reserved.

[ESP+12] = Environment data object address.

[ESP+16] = Command line linear address in environment data object.

Library initialization registers are defined as follows.

EIP = Library entry address.

ESP = User program stack.

CS = Code selector for base of linear address space.

DS = ES = SS = Data selector for base of linear address space.

FS = Data selector of base of Thread Information Block (TIB).

GS = 0.

EAX = EBX = 0.

ECX = EDX = 0.

ESI = EDI = 0.

EBP = 0.

[ESP+0] = Return address to system, (EAX) = return code.

[ESP+4] = Module handle for library module.

[ESP+8] = 0 (Initialization)

Note that a 32-bit library may specify that its entry address is in a 16-bit code object. In this case, the entry registers are the same as for entry to a library using the Segmented EXE format. These are documented elsewhere. This means that a 16-bit library may be relinked to take advantage of the benefits of the Linear EXE format (notably, efficient paging).

Library termination registers are defined as follows.

EIP = Library entry address.

ESP = User program stack.

CS = Code selector for base of linear address space.

DS = ES = SS = Data selector for base of linear address space.

FS = Data selector of base of Thread Information Block (TIB).

GS = 0.

EAX = EBX = 0.

ECX = EDX = 0.

ESI = EDI = 0.

EBP = 0.

[ESP+0] = Return address to system.

[ESP+4] = Module handle for library module.

[ESP+8] = 1 (Termination)

Note that Library termination is not allowed for libraries with 16-bit entries.

Object Table

The number of entries in the Object Table is given by the # Objects in Module field in the linear EXE header. Entries in the Object Table are numbered starting from one.

Each Object Table entry has the following format:

00h	VIRTUAL SIZE	RELOC BASE ADDR
08h	OBJECT FLAGS	PAGE TABLE INDEX
10h	# PAGE TABLE ENTRIES	RESERVED

Figure 43. Object Table

VIRTUAL SIZE = DD Virtual memory size.

This is the size of the object that will be allocated when the object is loaded. The object data length must be less than or equal to the total size of the pages in the EXE file for the object. This memory size must also be large enough to contain all of the iterated data and uninitialized data in the EXE file.

RELOC BASE ADDR = DD Relocation Base Address.

The relocation base address the object is currently relocated to. If the internal relocation fixups for the module have been removed, this is the address the object will be allocated at by the loader.

OBJECT FLAGS = DW Flag bits for the object.

The object flag bits have the following definitions.

0001h = Readable Object.
0002h = Writable Object.
0004h = Executable Object.

The readable, writable and executable flags provide support for all possible protections. In systems where all of these protections are not supported, the loader will be responsible for making the appropriate protection match for the system.

0008h = Resource Object.
0010h = Discardable Object.
0020h = Object is Shared.
0040h = Object has Preload Pages.
0080h = Object has Invalid Pages.
0100h = Object has Zero Filled Pages.
0200h = Object is Resident (valid for VDDs, PDDs only).
0300h = Object is Resident & Contiguous (VDDs, PDDs only).
0400h = Object is Resident & 'long-lockable' (VDDs, PDDs only).
0800h = Reserved for system use.
1000h = 16:16 Alias Required (80x86 Specific).
2000h = Big/Default Bit Setting (80x86 Specific).
The 'big/default' bit , for data segments, controls the setting of the Big bit in the segment descriptor. (The Big bit, or B-bit, determines whether ESP or SP is used as the stack pointer.) For code segments, this bit controls the setting of the Default bit in the segment descriptor. (The Default bit, or D-bit, determines whether the default word size is 32-bits or 16-bits. It also affects the interpretation of the instruction stream.)

4000h = Object is conforming for code (80x86 Specific).
8000h = Object I/O privilege level (80x86 Specific).
Only used for 16:16 Alias Objects.

PAGE TABLE INDEX = DD Object Page Table Index.

This specifies the number of the first object page table entry for this object. The object page table specifies where in the EXE file a page can be found for a given object and specifies per-page attributes.

The object table entries are ordered by logical page in the object table. In other words the object table entries are sorted based on the object page table index value.

PAGE TABLE ENTRIES = DD # of object page table entries for this object.

Any logical pages at the end of an object that do not have an entry in the object page table associated with them are handled as zero filled or invalid pages by the loader.

When the last logical pages of an object are not specified with an object page table entry, they are treated as either zero filled pages or invalid pages based on the last entry in the object page table for that object. If the last entry was neither a zero filled or invalid page, then the additional pages are treated as zero filled pages.

RESERVED = DD Reserved for future use. Must be set to zero.

Object Page Table

The Object page table provides information about a logical page in an object. A logical page may be an enumerated page, a pseudo page or an iterated page. The structure of the object page table in conjunction with the structure of the object table allows for efficient access of a page when a page fault occurs, while still allowing the physical page data to be located in the preload page, demand load page or iterated data page sections in the linear EXE module. The logical page entries in the Object Page Table are numbered starting from one. The Object Page Table is parallel to the Fixup Page Table as they are both indexed by the logical page number.

Each Object Page Table entry has the following format:

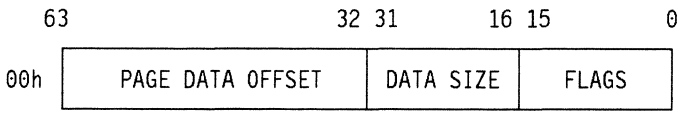


Figure 44. Object Page Table Entry

PAGE DATA OFFSET = DD Offset to the page data in the EXE file.

This field, when bit shifted left by the PAGE OFFSET SHIFT from the module header, specifies the offset from the beginning of the Preload Page section of the physical page data in the EXE file that corresponds to this logical page entry. The page data may reside in the Preload Pages, Demand Load Pages or the Iterated Data Pages sections.

A page might not start at the next available alignment boundary. Extra padding is acceptable between pages as long as each page starts on an alignment boundary. For example, several alignment boundarys may be skipped in order to start a frequently accessed page on a sector boundary.

If the FLAGS field specifies that this is a Zero-Filled page then the PAGE DATA OFFSET field will contain a 0.

If the logical page is specified as an iterated data page, as indicated by the FLAGS field, then this field specifies the offset into the Iterated Data Pages section.

The logical page number (Object Page Table index), is used to index the Fixup Page Table to find any fixups associated with the logical page.

DATA SIZE = DW Number of bytes of data for this page.

This field specifies the actual number of bytes that represent the page in the file. If the PAGE SIZE field from the module header is greater than the value of this field and the FLAGS field indicates a Legal Physical Page, the remaining bytes are to be filled with zeros. If the FLAGS field indicates an Iterated Data Page, the iterated data records will completely fill out the remainder.

FLAGS = DW Attributes specifying characteristics of this logical page.

The bit definitions for this word field follow,

00h = Legal Physical Page in the module (Offset from Preload Page Section).
01h = Iterated Data Page (Offset from Iterated Data Pages Section).
02h = Invalid Page (zero).
03h = Zero Filled Page (zero).
04h = Range of Pages.
05h = Compressed Page (Offset from Preload Pages Section).

Resource Table

The resource table is an array of resource table entries. Each resource table entry contains a type ID and name ID. These entries are used to locate resource objects contained in the Object table. The number of entries in the resource table is defined by the Resource Table Count located in the linear EXE header. More than one resource may be contained within a single object. Resource table entries are in a sorted order, (ascending, by Resource Name ID within the Resource Type ID). This allows the DosGetResource API function to use a binary search when looking up a resource in a 32-bit module instead of the linear search being used in the current 16-bit module.

Each resource entry has the following format:

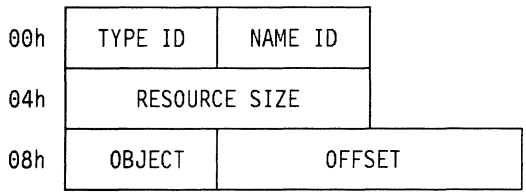


Figure 45. Resource Table

TYPE ID = DW Resource type ID.

The type of resources are:

01h = RT_POINTER = mouse pointer shape
02h = RT_BITMAP = bitmap
03h = RT_MENU = menu template
04h = RT_DIALOG = dialog template
05h = RT_STRING = string tables
06h = RT_FONTDIR = font directory
07h = RT_FONT = font
08h = RT_ACCELTABLE = accelerator tables
09h = RT_RCDATA = binary data
0Ah = RT_MESSAGE = error msg tables
0Bh = RT_DLGINCLUDE = dialog include file name
0Ch = RT_VKEYTBL = key to vkey tables
0Dh = RT_KEYTBL = key to UGL tables
0Eh = RT_CHARTBL = glyph to character tables
0Fh = RT_DISPLAYINFO = screen display information
10h = RT_FKASHORT = function key area short form
11h = RT_FKALONG = function key area long form
12h = RT_HELPTABLE = Help table for Cary Help manager
13h = RT_HELPSUBTABLE = Help subtable for Cary Help manager
14h = RT_FDDIR = DBCS uniq/font driver directory
15h = RT_FD = DBCS uniq/font driver

NAME ID = DW An ID used as a name for the resource when referred to.

RESOURCE SIZE = DD The number of bytes the resource consists of.

OBJECT = DW The number of the object which contains the resource.

OFFSET = DD The offset within the specified object where the resource begins.

Resident or Non-resident Name Table Entry

The resident and non-resident name tables define the ASCII names and ordinal numbers for exported entries in the module. In addition the first entry in the resident name table contains the module name. These tables are used to translate a procedure name string into an ordinal number by searching for a matching name string. The ordinal number is used to locate the entry point information in the entry table.

The resident name table is kept resident in system memory while the module is loaded. It is intended to contain the exported entry point names that are frequently dynamicaly linked to by name. Non-resident names are not kept in memory and are read from the EXE file when a dynamic link reference is made. Exported entry point names that are infrequently dynamicaly linked to by name or are commonly referenced by ordinal number should be placed in the non-resident name table. The trade off made for references by name is performance vs memory usage.

Import references by name require these tables to be searched to obtain the entry point ordinal number. Import references by ordinal number provide the fastest lookup since the search of these tables is not required.

Installable File Systems, Physical Device Drivers, and Virtual Device Drivers are closed after the file is loaded. Any refeference to the non-resident name table after this time will fail.

The strings are CASE SENSITIVE and are NOT NULL TERMINATED.

Each name table entry has the following format:

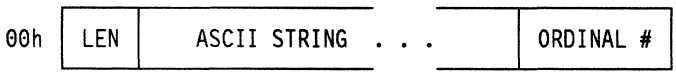


Figure 46. Resident or Non-resident Name Table Entry

LEN = DB String Length.

This defines the length of the string in bytes. A zero length indicates there are no more entries in table. The length of each ascii name string is limited to 255 characters.

The high bit in the LEN field (bit 7) is defined as an Overload bit. This bit signifies that additional information is contained in the linear EXE module and will be used in the future for parameter type checking.

ASCII STRING = DB ASCII String.

This is a variable length string with it's length defined in bytes by the LEN field. The string is case case sensitive and is not null terminated.

ORDINAL # = DW Ordinal number.

The ordinal number in an ordered index into the entry table for this entry point.

Entry Table

The entry table contains object and offset information that is used to resolve fixup references to the entry points within this module. Not all entry points in the entry table will be exported, some entry points will only be used within the module. An ordinal number is used to index into the entry table. The entry table entries are numbered starting from one.

The list of entries are compressed into 'bundles', where possible. The entries within each bundle are all the same size. A bundle starts with a count field which indicates the number of entries in the bundle. The count is followed by a type field which identifies the bundle format. This provides both a means for saving space as well as a mechanism for extending the bundle types.

The type field allows the definition of 256 bundle types. The following bundle types will initially be defined:

Unused Entry.
16-bit Entry.
286 Call Gate Entry.
32-bit Entry.
Forwarder Entry.

The bundled entry table has the following format:

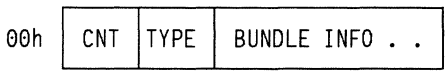


Figure 47. Entry Table

CNT = DB Number of entries.

This is the number of entries in this bundle.

A zero value for the number of entries identifies the end of the entry table. There is no further bundle information when the number of entries is zero. In other words the entry table is terminated by a single zero byte.

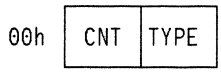
TYPE = DB Bundle type.

This defines the bundle type which determines the contents of the BUNDLE INFO.

The follow types are defined:

00h = Unused Entry.
01h = 16-bit Entry.
02h = 286 Call Gate Entry.
03h = 32-bit Entry.
04h = Forwarder Entry.
80h = Parameter Typing Information Present.
This bit signifies that additional information is contained in the linear EXE module and will be used in the future for parameter type checking.

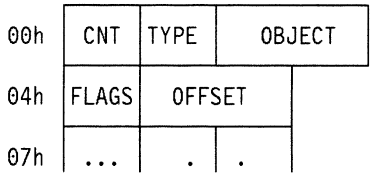
The following is the format for each bundle type:



CNT = DB Number of entries.

This is the number of unused entries to skip.

TYPE = DB 0 (Unused Entry)



CNT = DB Number of entries.

This is the number of 16-bit entries in this bundle. The flags and offset value are repeated this number of times.

TYPE = DB 1 (16-bit Entry)

OBJECT = DW Object number.

This is the object number for the entries in this bundle.

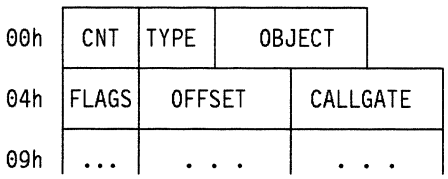
FLAGS = DB Entry flags.

These are the flags for this entry point. They have the following definition.

01h = Exported entry flag.
F8h = Parameter word count mask.

OFFSET = DW Offset in object.

This is the offset in the object for the entry point defined at this ordinal number.



CNT = DB Number of entries.

This is the number of 286 call gate entries in this bundle. The flags, callgate, and offset value are repeated this number of times.

TYPE = DB 2 (286 Call Gate Entry)

The 286 Call Gate Entry Point type is needed by the loader only if ring 2 segments are to be supported. 286 Call Gate entries contain 2 extra bytes which are used by the loader to store an LDT callgate selector value.

OBJECT = DW Object number.

This is the object number for the entries in this bundle.

FLAGS = DB Entry flags.

These are the flags for this entry point. They have the following definition.

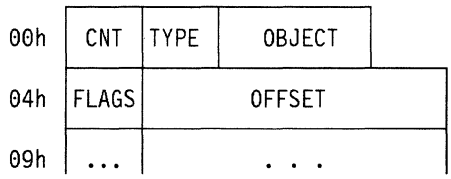
01h = Exported entry flag.
F8h = Parameter word count mask.

OFFSET = DW Offset in object.

This is the offset in the object for the entry point defined at this ordinal number.

CALLGATE = DW Callgate selector.

The callgate selector is a reserved field used by the loader to store a call gate selector value for references to ring 2 entry points. When a ring 3 reference to a ring 2 entry point is made, the callgate selector with a zero offset is place in the relocation fixup address. The segment number and offset in segment is placed in the LDT callgate.



CNT = DB Number of entries.

This is the number of 32-bit entries in this bundle. The flags and offset value are repeated this number of times.

TYPE = DB 3 (32-bit Entry)

The 32-bit Entry type will only be defined by the linker when the offset in the object can not be specified by a 16-bit offset.

OBJECT = DW Object number.

This is the object number for the entries in this bundle.

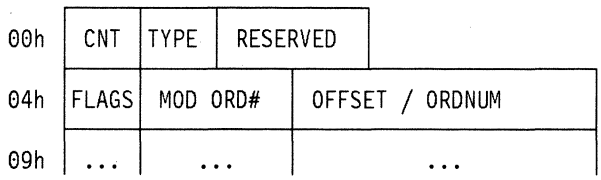
FLAGS = DB Entry flags.

These are the flags for this entry point. They have the following definition.

01h = Exported entry flag.
F8h = Parameter dword count mask.

OFFSET = DD Offset in object.

This is the offset in the object for the entry point defined at this ordinal number.



CNT = DB Number of entries.

This is the number of forwarder entries in this bundle. The FLAGS, MOD ORD#, and OFFSET/ORDNUM values are repeated this number of times.

TYPE = DB 4 (Forwarder Entry)

RESERVED = DW 0

This field is reserved for future use.

FLAGS = DB Forwarder flags.

These are the flags for this entry point. They have the following definition.

01h = Import by ordinal.
F7h = Reserved for future use; should be zero.

MOD ORD# = DW Module Ordinal Number

This is the index into the Import Module Name Table for this forwarder.

OFFSET / ORDNUM = DD Procedure Name Offset or Import Ordinal Number

If the FLAGS field indicates import by ordinal, then this field is the ordinal number into the Entry Table of the target module, otherwise this field is the offset into the Procedure Names Table of the target module.

A Forwarder entry (type = 4) is an entry point whose value is an imported reference. When a load time fixup occurs whose target is a forwarder, the loader obtains the address imported by the forwarder and uses that imported address to resolve the fixup.

A forwarder may refer to an entry point in another module which is itself a forwarder, so there can be a chain of forwarders. The loader will traverse the chain until it finds a non-forwarded entry point which terminates the chain , and use this to resolve the original fixup. Circular chains are detected by the loader and result in a load time error. A maximum of 1024 forwarders is allowed in a chain; more than this results in a load time error.

Forwarders are useful for merging and recombining API calls into different sets of libraries, while maintaining compatibility with applications. For example, if one wanted to combine MONCALLS, MOUCALLS, and VIOCALLS into a single libraries, one could provide entry points for the three libraries that are forwarders pointing to the common implementation.

Module Format Directives Table

The Module Format Directives Table is an optional table that allows additional options to be specified. It also allows for the extension of the linear EXE format by allowing additional tables of information to be added to the linear EXE module without affecting the format of the linear EXE header. Likewise, module format directives provide a place in the linear EXE module for 'temporary tables' of information, such as incremental linking information and statistic informa-

tion gathered on the module. When there are no module format directives for a linear EXE module, the fields in the linear EXE header referencing the module format directives table are zero.

Each Module Format Directive Table entry has the following format:

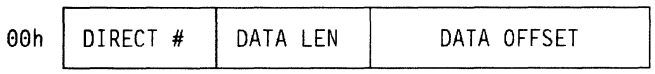


Figure 48. Module Format Directive Table

DIRECT # = DW Directive number.

The directive number specifies the type of directive defined. This can be used to determine the format of the information in the directive data. The following directive numbers have been defined:

8000h = Resident Flag Mask.
Directive numbers with this bit set indicate that the directive data is in the resident area and will be kept resident in memory when the module is loaded.

8001h = Verify Record Directive. (Verify record is a resident table.)
0002h = Language Information Directive. (This is a non-resident table.)
0003h = Co-Processor Required Support Table.
0004h = Thread State Initialization Directive.
0005h = C Set + + Browse Information.

Additional directives can be added as needed in the future, as long as they do not overlap previously defined directive numbers.

DATA LEN = DW Directive data length.

This specifies the length in byte of the directive data for this directive number.

DIRECTIVE OFFSET = DD Directive data offset.

This is the offset to the directive data for this directive number. It is relative to beginning of linear EXE header for a resident table, and relative to the beginning of the EXE file for non-resident tables.

Verify Record Directive Table

The Verify Record Directive Table is an optional table. It maintains a record of the pages in the EXE file that have been fixed up and written back to the original linear EXE module, along with the module dependencies used to perform these fixups. This table provides an efficient means for verifying the virtual addresses required for the fixed up pages when the module is loaded.

Each Verify Record entry has the following format:

00h	# OF ENTRY		
02h	MOD ORD #	VERSION	MOD # OBJ
08h	OBJECT #	BASE ADDR	VIRTUAL
0Eh	. . .	. . .	. . .

Figure 49. Verify Record Table

OF ENTRY = DW Number of module dependencies.

This field specifies how many entries there are in the verify record directive table. This is equal to the number of modules referenced by this module.

MOD ORD # = DW Ordinal index into the Import Module Name Table.

This value is an ordered index in to the Import Module Name Table for the referenced module.

VERSION = DW Module Version.

This is the version of the referenced module that the fixups were originally performed. This is used to insure the same version of the referenced module is loaded that was fixed up in this module and therefore the fixups are still correct. This requires the version number in a module to be incremented anytime the entry point offsets change.

MOD # OBJ = DW Module # of Object Entries.

This field is used to identify the number of object verify entries that follow for the referenced module.

OBJECT # = DW Object # in Module.

This field specifies the object number in the referenced module that is being verified.

BASE ADDR = DW Object load base address.

This is the address that the object was loaded at when the fixups were performed.

VIRTUAL = DW Object virtual address size.

This field specifies the total amount of virtual memory required for this object.

Per-Page Checksum

The Per-Page Checksum table provides space for a cryptographic checksum for each physical page in the EXE file.

The checksum table is arranged such that the first entry in the table corresponds to the first logical page of code/data in the EXE file (usually a preload page) and the last entry corresponds to the last logical page in the EXE file (usually a iterated data page).

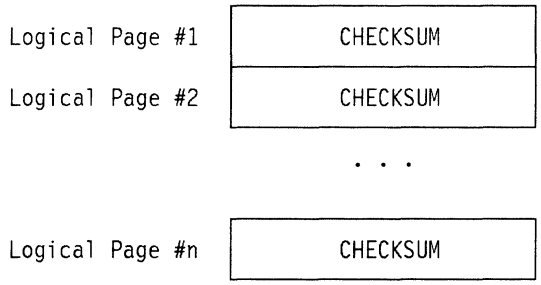


Figure 50. Per-Page Checksum

CHECKSUM = DD Cryptographic checksum.

Fixup Page Table

The Fixup Page Table provides a simple mapping of a logical page number to an offset into the Fixup Record Table for that page.

This table is parallel to the Object Page Table, except that there is one additional entry in this table to indicate the end of the Fixup Record Table.

The format of each entry is:

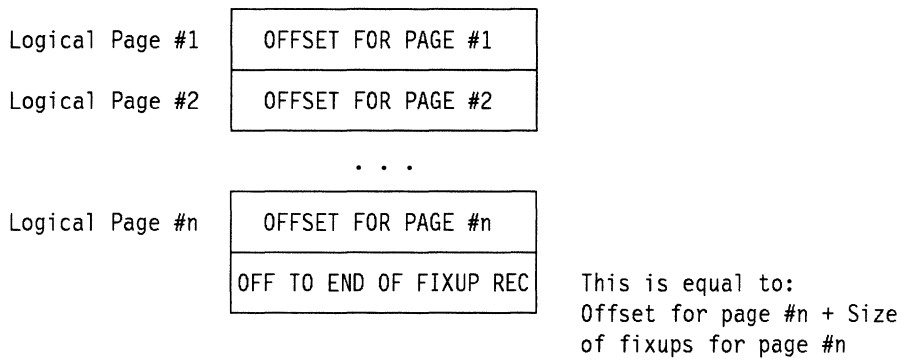


Figure 51. Fixup Page Table

OFFSET FOR PAGE # = DD Offset for fixup record for this page.

This field specifies the offset, from the beginning of the fixup record table, to the first fixup record for this page.

OFF TO END OF FIXUP REC = DD Offset to the end of the fixup records.

This field specifies the offset following the last fixup record in the fixup record table. This is the last entry in the fixup page table.

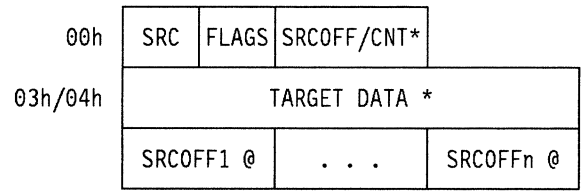
The fixup records are kept in order by logical page in the fixup record table. This allows the end of each page's fixup records is defined by the offset for the next logical page's

fixup records. This last entry provides support of this mechanism for the last page in the fixup page table.

Fixup Record Table

The Fixup Record Table contains entries for all fixups in the linear EXE module. The fixup records for a logical page are grouped together and kept in sorted order by logical page number. The fixups for each page are further sorted such that all external fixups and internal selector/pointer fixups come before internal non-selector/non-pointer fixups. This allows the loader to ignore internal fixups if the loader is able to load all objects at the addresses specified in the object table.

Each relocation record has the following format:




```
* These fields are variable size.
@ These fields are optional.
```

Figure 52. Fixup Record Table

SRC = DB Source type.

The source type specifies the size and type of the fixup to be performed on the fixup source. The source type is defined as follows:

0Fh = Source mask.
00h = Byte fixup (8-bits).
01h = (undefined).
02h = 16-bit Selector fixup (16-bits).
03h = 16:16 Pointer fixup (32-bits).
04h = (undefined).
05h = 16-bit Offset fixup (16-bits).
06h = 16:32 Pointer fixup (48-bits).
07h = 32-bit Offset fixup (32-bits).
08h = 32-bit Self-relative offset fixup (32-bits).
10h = Fixup to Alias Flag.
When the 'Fixup to Alias' Flag is set, the source fixup refers to the 16:16 alias for the object. This is only valid for source types of 2, 3, and 6. For fixups such as this, the linker and loader will be required to perform additional checks such as ensuring that the target offset for this fixup is less than 64K.

20h = Source List Flag.
When the 'Source List' Flag is set, the SRCOFF field is compressed to a byte and contains the number of source offsets, and a list of source offsets follows the end of fixup record (after the optional additive value).

FLAGS = DB Target Flags.

The target flags specify how the target information is interpreted. The target flags are defined as follows:

03h = Fixup target type mask.
00h = Internal reference.
01h = Imported reference by ordinal.
02h = Imported reference by name.
03h = Internal reference via entry table.
04h = Additive Fixup Flag.
When set, an additive value trails the fixup record (before the optional source offset list).

08h = Reserved. Must be zero.
10h = 32-bit Target Offset Flag.
When set, the target offset is 32-bits, otherwise it is 16-bits.

20h = 32-bit Additive Fixup Flag.
When set, the additive value is 32-bits, otherwise it is 16-bits.

40h = 16-bit Object Number/Module Ordinal Flag.
When set, the object number or module ordinal number is 16-bits, otherwise it is 8-bits.

80h = 8-bit Ordinal Flag.
When set, the ordinal number is 8-bits, otherwise it is 16-bits.

SRCOFF = DW/CNT = DB Source offset or source offset list count.

This field contains either an offset or a count depending on the Source List Flag. If the Source List Flag is set, a list of source offsets follows the additive field and this field contains the count of the entries in the source offset list. Otherwise, this is the single source offset for the fixup. Source offsets are relative to the beginning of the page where the fixup is to be made.

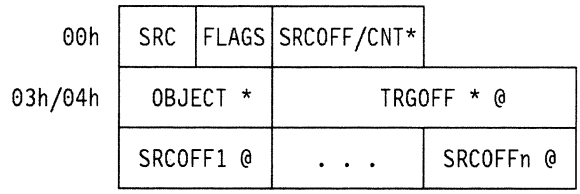
Note that for fixups that cross page boundaries, a separate fixup record is specified for each page. An offset is still used for the 2nd page but it now becomes a negative offset since the fixup originated on the preceding page. (For example, if only the last one byte of a 32-bit address is on the page to be fixed up, then the offset would have a value of -3.)

TARGET DATA = Target data for fixup.

The format of the TARGET DATA is dependent upon target flags.

SRCOFF1 - SRCOFFn = DW[] Source offset list.

This list is present if the Source List Flag is set in the Target Flags field. The number of entries in the source offset list is defined in the SRCOFF/CNT field. The source offsets are relative to the beginning of the page where the fixups are to be made.



* These fields are variable size.
@ These fields are optional.

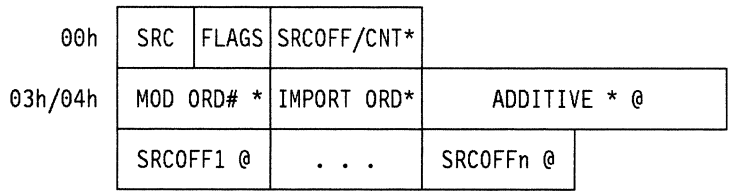
Figure 53. Internal Fixup Record

OBJECT = D[B|W] Target object number.

This field is an index into the current module's Object Table to specify the target Object. It is a Byte value when the '16-bit Object Number/Module Ordinal Flag' bit in the target flags field is clear and a Word value when the bit is set.

TRGOFF = D[W|D] Target offset.

This field is an offset into the specified target Object. It is not present when the Source Type specifies a 16-bit Selector fixup. It is a Word value when the '32-bit Target Offset Flag' bit in the target flags field is clear and a Dword value when the bit is set.



* These fields are variable size.
@ These fields are optional.

Figure 54. Import by Ordinal Fixup Record

MOD ORD # = D[B|W] Ordinal index into the Import Module Name Table.

This value is an ordered index in to the Import Module Name Table for the module containing the procedure entry point. It is a Byte value when the '16-bit Object Number/Module Ordinal' Flag bit in the target flags field is clear and a Word value when the bit is set. The loader creates a table of pointers with each pointer in the table corresponds to the modules named in the Import Module Name Table. This value is used by the loader to index into this table created by the loader to locate the referenced module.

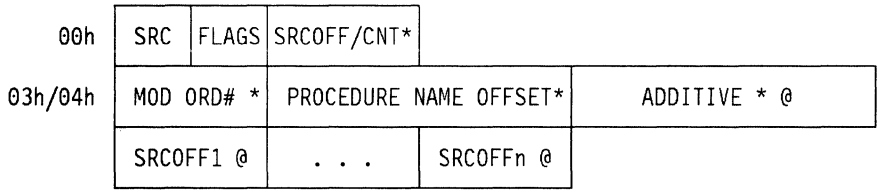
IMPORT ORD = D[B|W|D] Imported ordinal number.

This is the imported procedure's ordinal number. It is a Byte value when the '8-bit Ordinal' bit in the target flags field is set. Otherwise it is a Word value when the '32-bit Target Offset Flag' bit in the target flags field is clear and a Dword value when the bit is set.

ADDITIVE = D[W|D] Additive fixup value.

This field exists in the fixup record only when the 'Additive Fixup Flag' bit in the target flags field is set. When the 'Additive Fixup Flag' is clear the fixup record does not contain this field and is immediately followed by the next fixup record (or by the source offset list for this fixup record).

This value is added to the address derived from the target entry point. This field is a Word value when the '32-bit Additive Flag' bit in the target flags field is clear and a Dword value when the bit is set.



* These fields are variable size.
@ These fields are optional.

Figure 55. Import by Name Fixup Record

MOD ORD # = D[B|W] Ordinal index into the Import Module Name Table.

This value is an ordered index in to the Import Module Name Table for the module containing the procedure entry point. It is a Byte value when the '16-bit Object Number/Module Ordinal' Flag bit in the target flags field is clear and a Word value when the bit is set. The loader creates a table of pointers with each pointer in the table corresponds to the modules named in the Import Module Name Table. This value is used by the loader to index into this table created by the loader to locate the referenced module.

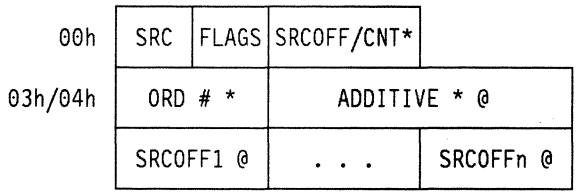
PROCEDURE NAME OFFSET = D[W|D] Offset into the Import Procedure Name Table.

This field is an offset into the Import Procedure Name Table. It is a Word value when the '32-bit Target Offset Flag' bit in the target flags field is clear and a Dword value when the bit is set.

ADDITIVE = D[W|D] Additive fixup value.

This field exists in the fixup record only when the 'Additive Fixup Flag' bit in the target flags field is set. When the 'Additive Fixup Flag' is clear the fixup record does not contain this field and is immediately followed by the next fixup record (or by the source offset list for this fixup record).

This value is added to the address derived from the target entry point. This field is a Word value when the '32-bit Additive Flag' bit in the target flags field is clear and a Dword value when the bit is set.



* These fields are variable size.
@ These fields are optional.

Figure 56. Internal Entry Table Fixup Record

ENTRY # = D[B|W] Ordinal index into the Entry Table.

This field is an index into the current module's Entry Table to specify the target Object and offset. It is a Byte value when the '16-bit Object Number/Module Ordinal' Flag bit in the target flags field is clear and a Word value when the bit is set.

ADDITIVE = D[W|D] Additive fixup value.

This field exists in the fixup record only when the 'Additive Fixup Flag' bit in the target flags field is set. When the 'Additive Fixup Flag' is clear the fixup record does not contain this field and is immediately followed by the next fixup record (or by the source offset list for this fixup record).

This value is added to the address derived from the target entry point. This field is a Word value when the '32-bit Additive Flag' bit in the target flags field is clear and a Dword value when the bit is set.

Import Module Name Table

The import module name table defines the module name strings imported through dynamic link references. These strings are referenced through the imported relocation fixups.

To determine the length of the import module name table subtract the import module name table offset from the import procedure name table offset. These values are located in the linear EXE header. The end of the import module name table is not terminated by a special character, it is followed directly by the import procedure name table.

The strings are CASE SENSITIVE and NOT NULL TERMINATED.

Each name table entry has the following format:

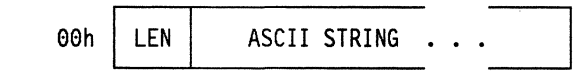


Figure 57. Import Module Name Table

LEN = DB String Length.

This defines the length of the string in bytes. The length of each ascii name string is limited to 255 characters.

ASCII STRING = DB ASCII String.

This is a variable length string with it's length defined in bytes by the LEN field. The string is case sensitive and is not null terminated.

Import Procedure Name Table

The import procedure name table defines the procedure name strings imported by this module through dynamic link references. These strings are referenced through the imported relocation fixups.

To determine the length of the import procedure name table add the fixup section size to the fixup page table offset, this computes the offset to the end of the fixup section, then subtract the import procedure name table offset. These values are located in the linear EXE header. The import procedure name table is followed by the data pages section. Since the data pages section is aligned on a 'page size' boundary, padded space may exist between the last import name string and the first page in the data pages section. If this padded space exists it will be zero filled.

The strings are CASE SENSITIVE and NOT NULL TERMINATED.

Each name table entry has the following format:

Figure 58. Import Procedure Name Table

LEN = DB String Length.

This defines the length of the string in bytes. The length of each ascii name string is limited to 255 characters.

The high bit in the LEN field (bit 7) is defined as an Overload bit. This bit signifies that additional information is contained in the linear EXE module and will be used in the future for parameter type checking.

ASCII STRING = DB ASCII String.

This is a variable length string with it's length defined in bytes by the LEN field. The string is case sensitive and is not null terminated.

Note: The first entry in the import procedure name table must be a null entry. That is, the LEN field should be zero followed an empty ASCII STRING (no bytes).

Preload Pages

The Preload Pages section is an optional section in the linear EXE module that coalesces a 'preload page set' into a contiguous section. The preload page set can be defined as the set of first used pages in the module. The preload page set can be specified by the application developer or can be derived by a tool that analyzes the programs memory usage while it is running. By grouping the preload page set together, the preload pages can be read from the linear EXE module with one disk read.

The structure of the preload pages is no different than if they were demand loaded. Their sizes are determined by the Object Page Table entries that correspond. If the specified size is less than the PAGE SIZE field given in the linear EXE module header the remainder of the page is filled with zeros when loaded.

All pages begin on a PAGE OFFSET SHIFT boundary from the base of the preload page section, as specified in the linear EXE header. The pages are ordered by logical page number within this section.

Note that OS/2 2.x does not respect the preload of pages as specified in the executable file for performance reasons.

Demand Load Pages

The Demand Loaded Pages section contains all the non-iterated pages for a linear EXE module that are not preloaded. When required, the whole page is loaded into memory from the module. The characteristics of each of these pages is specified in the Object Page Table. Every page begins on a PAGE OFFSET SHIFT boundary aligned offset from the demand loaded pages base specified in the linear EXE header. Their sizes are determined by the Object Page Table entries that correspond. If the specified size is less than the PAGE SIZE field given in the linear EXE module header the remainder of the page is filled with zeros when loaded. The pages are ordered by logical page number within this section.

Iterated Data Pages

The Iterated Data Pages section contains all the pages for a linear EXE module that are iterated. When required, the set of iteration records are loaded into memory from the module and expanded to reconstitute the page. Every set of iteration records begins on a PAGE OFFSET SHIFT offset from the OBJECT ITER PAGES OFF specified in the linear EXE header. Their sizes are determined by the Object Page Table entries that correspond. The pages are ordered by logical page number within this section.

This record structure is used to describe the iterated data for an object on a per-page basis.

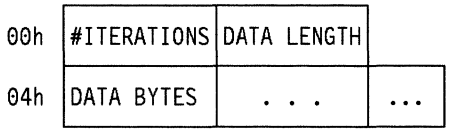


Figure 59. Object Iterated Data Record (Iteration Record)

#ITERATIONS = DW Number of iterations.

This specifies the number of times that the data is replicated.

DATA LENGTH = DW The size of the data pattern in bytes.

This specifies the number of bytes of data of which the pattern consists. The maximum size is one half of the PAGE SIZE (given in the module header). If a pattern exceeds this value then the data page will not be condensed into iterated data.

DATA = DB * DATA LENGTH The Data pattern to be replicated.

The next iteration record will immediately follow the last byte of the pattern. The offset of the next iteration record is easily calculated from the offset of this record by adding the DATA LENGTH field and the sizes of the #ITERATIONS and DATA LENGTH fields.

Debug Information

The debug information is defined by the debugger and is not controlled by the linear EXE format or linker. The only data defined by the linear EXE format relative to the debug information is it's offset in the EXE file and length in bytes as defined in the linear EXE header.

To support multiple debuggers the first word of the debug information is a type field which determines the format of the debug information.

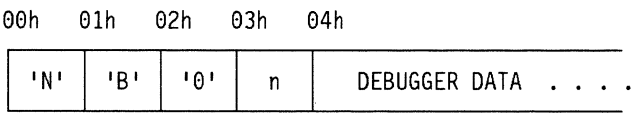


Figure 60. Debug Information

TYPE = DB DUP 4 Format type.

This defines the type of debugger data that exists in the remainder of the debug information. The signature consists of a string of four (4) ASCII characters: "NB0" followed by the ASCII representation for 'n'. The values for 'n' are defined as follows.

These format types are defined.

00h = 32-bit CodeView debugger format.
01h = AIX debugger format.
02h = 16-bit CodeView debugger format.
04h = 32-bit OS/2 PM debugger (IBM) format.

DEBUGGER DATA = Debugger specific data.

The format of the debugger data is defined by the debugger that is being used.

The values defined for the type field are not enforced by the system. It is the responsibility of the linker or debugging tools to follow the convention for the type field that is defined here.