OMF: Relocatable Object Module Format

Table of Contents

The Object Record Format

Record Format

All object records conform to the following format:

<----------Record Length in Bytes---------->

1	2	<variable>	1
Record Type	Record Length	Record Contents	Checksum or 0

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. An odd Record Type indicates that certain numeric fields within the record contain 32-bit values; an even Record Type indicates that those fields contain 16-bit values. The affected fields are described with each record. Note that this principle does not govern the Use32/Use16 segment attribute (which is set in the ACBP byte of SEGDEF records); it simply specifies the size of certain numeric fields within the record. It is possible to use 16-bit OMF records to generate 32-bit segments, or vice versa.

The Record Length field is a 2-byte field that gives the length of the remainder of the object record in bytes (excluding the bytes in the Record Type and Record Length fields). The record length is stored with the low-order byte first. An entire record occupies 3 bytes plus the number of bytes in the Record Length field.

The Record Contents field varies in size and format, depending on the record type.

The Checksum field is a 1-byte field that contains the negative sum (modulo 256) of all other bytes in the record. In other words, the checksum byte is calculated so that the low-order byte of the sum of all the bytes in the record, including the checksum byte, equals 0. Overflow is ignored. Some compilers write a 0 byte rather than computing the checksum, so either form should be accepted by programs that process object modules.

> ***Note:*** *The maximum size of the entire record (unless otherwise noted for specific record types) is 1024 bytes.*

Frequent Object Record Subfields

The contents of each record are determined by the record type, but certain subfields appear frequently enough to be explained separately. The format of such fields is below.

Names

A name string is encoded as an 8-bit unsigned count followed by a string of count characters, refered to as *count, char* format. The character set is usually some ASCII subset. A null name is specified by a single byte of 0 (indicating a string of length 0).

Indexed References

Certain items are ordered by occurrence and are referenced by index. The first occurrence of the item has index number 1. Index fields may contain 0 (indicating that they are not present) or values from 1 through 7FFF. The index number field in an object record can be either 1 or 2 bytes long. If the number is in the range 0–7FH, 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. The code to decode an index is:

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

Type Indexes

Type Index fields occupy 1 or 2 bytes and occur in PUBDEF, LPUBDEF, COMDEF, LCOMDEF, EXTDEF, and LEXTDEF records. These type index fields were used in old versions of the OMF to reference TYPDEF records. This usage is obsolete. This field is usually left empty (encoded as 1 byte with value 0). However some linkers may use this for debug information or other purposes.

Ordered Collections

Certain records and record groups are ordered so that the records may be referred to with indexes (the format of indexes is described in the "Indexed References" section). The same format is used whether an index refers to names, logical segments, or other items.

The overall ordering is obtained from the order of the records 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 not specified).

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, and so forth, with the highest index for the last name in the last LNAMES record encountered.

The ordered collections are:

Names	Ordered by occurrence of LNAMES records and names within each. Referenced as a name index.
Logical Segments	Ordered by occurrence of SEGDEF records in file. Referenced as a segment index.
Groups	Ordered by occurrence of GRPDEF records in file. Referenced as a group index.
External Symbols	Ordered by occurrence of EXTDEF, COMDEF, LEXTDEF, and LCOMDEF records and symbols within each. Referenced as an external name index (in FIXUP subrecords).

Numeric 2-Byte and 4-Byte Fields

Words and double words (16- and 32-bit quantities, respectively) are stored in little endian byte order (lowest address is least significant).

Certain records, notably SEGDEF, PUBDEF, LPUBDEF, LINNUM, LEDATA, LIDATA, FIXUPP, and MODEND, contain size, offset, and displacement values that 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 (that is, the 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. The values are zero-extended when applied to Use32 segments.

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

Order of Records

The sequence in which the types of object records appear in an object module is fairly flexible in some respects. Several record types are optional, and if the type of information they carry is unnecessary, they are omitted from the object module. In addition, most object record types can occur more than once in the same object module. And because object records are variable in length, it is often possible to choose between combining information into one large record or breaking it down into several smaller records of the same type.

An important constraint on the order in which object records appear is the need for some types of object records to refer to information contained in other records. Because the linker processes the records sequentially, object records containing such information must precede the records that refer to the information. For example, two types of object records, SEGDEF and GRPDEF, refer to the names contained in an LNAMES record. Thus, an LNAMES record must appear before any SEGDEF or GRPDEF records that refer to it so that the names in the LNAMES record are known to the linker by the time it processes the SEGDEF or GRPDEF records.

The record order is chosen so that the number of linker passes through an object module are minimized. Most linkers make two passes through the object modules: the first pass may be cut short by the presence of the Link Pass Separator COMENT record; the second pass processes all records.

For greatest linking speed, all symbolic information should occur at the start of the object module. This order is recommended but not mandatory. The general ordering is:

Identifier Record(s)

THEADR or LHEADR record

***Note**: This must be the first record.*

Records Processed by Pass 1

The following records may occur in any order but they *must* precede the Link Pass Separator if it is present:

COMENT records identifying object format and extensions

COMENT records other than Link Pass Separator comment

LNAMES or LLNAMES records providing ordered name list

SEGDEF records providing ordered list of program segments

GRPDEF records providing ordered list of logical segments

TYPDEF records (obsolete)

ALIAS records

PUBDEF records locating and naming public symbols

LPUBDEF records locating and naming private symbols

COMDEF, LCOMDEF, EXTDEF, LEXTDEF, and CEXTDEF records

***Note:** This group of records is indexed together, so external name index fields in FIXUPP records may refer to any of the record types listed.*

Link Pass Separator (Optional)

COMENT class A2 record is used to indicate that Pass 1 of the linker is complete. When this record is encountered, many linkers stop their first pass over the object file. Records preceding the link pass separator define the symbolic information for the file.

For greater linking speed, all LIDATA, LEDATA, FIXUPP, BAKPAT, INCDEF, and LINNUM records should come after the A2 COMENT record, but this is not required. Pass 2 should begin again at the start of the object module so that these records are processed in Pass 2 regardless of where they are placed in the object module.

Records Ignored by Pass 1 and Processed by Pass 2

The following records may come before or after the Link Pass Separator:

LIDATA, LEDATA, or COMDAT records followed by applicable FIXUPP records

FIXUPP records containing only THREAD subrecords

BAKPAT and NBKPAT FIXUPP records

COMENT class A0, subrecord type 03 (INCDEF) records containing incremental compilation information for FIXUPP and LINNUM records

LINNUM and LINSYM records providing line number and program code or data association

Terminator

MODEND record indicating end of module with optional start address

Record Specifics

The following is a list of record types that have been implemented and are described within the body of this document. Details of each implemented record (form and content) are presented in the following sections. The records are listed sequentially by hex value. Conflicts between various OMFs that overlap in their use of record types or fields are marked. For information on obsolete records, please refer to Appendix 3.

Currently Implemented Records

80H	**THEADR**	**Translator Header Record**
82H	**LHEADR**	**Library Module Header Record**
88H	**COMENT**	**Comment Record (Including all comment class extensions)**
8AH/8BH	**MODEND**	**Module End Record**
8CH	**EXTDEF**	**External Names Definition Record**
90H/91H	**PUBDEF**	**Public Names Definition Record**
94H/95H	**LINNUM**	**Line Numbers Record**
96H	**LNAMES**	**List of Names Record**
98H/99H	**SEGDEF**	**Segment Definition Record**
9AH	**GRPDEF**	**Group Definition Record**
9CH/9DH	**FIXUPP**	**Fixup Record**
A0H/A1H	**LEDATA**	**Logical Enumerated Data Record**
A2H/A3H	**LIDATA**	**Logical Iterated Data Record**
B0H	**COMDEF**	**Communal Names Definition Record**
B2H/B3H	**BAKPAT**	**Backpatch Record**
B4H	**LEXTDEF**	**Local External Names Definition Record**
B6H/B7H	**LPUBDEF**	**Local Public Names Definition Record**
B8H	**LCOMDEF**	**Local Communal Names Definition Record**
BCH	**CEXTDEF**	**COMDAT External Names Definition Record**
C2H/C3H	**COMDAT**	**Initialized Communal Data Record**
C4H/C5H	**LINSYM**	**Symbol Line Numbers Record**
C6H	**ALIAS**	**Alias Definition Record**
C8H/C9H	**NBKPAT**	**Named Backpatch Record**
CAH	**LLNAMES**	**Local Logical Names Definition Record**
CCH	**VERNUM**	**OMF Version Number Record**
CEH	**VENDEXT**	**Vendor-specific OMF Extension Record**
F0H		**Library Header Record**
		Although this is not actually an OMF record type, the presence of a record with F0H as the first byte indicates that the module is a library. The format of a library file is given in Appendix 2.
F1H		**Library End Record**

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.

History

Unchanged from the original Intel 8086 specification.

Record Format

1	2	1	<-------String Length------>	1
80	Record Length	String Length	Name String	Checksum

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 file that contains a program's source code (if supplied by the language translator), or may be specified directly by the programmer (for example, TITLE pseudo-operand or assembler NAME directive).

Notes

The name string is always present; a null name is allowed but not recommended (because it doesn't provide much information for a debugging program).

The name string indicates the full path and filename of the file that contained the source code for the module.

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

Example

The following THEADR record was generated by the Microsoft C Compiler:

```
      0    1    2    3    4    5    6    7    8    9    A    B    C    D    E    F
0000  80   09   00   07   68   65   6C   6C   6F   2E   63   CB                   ...hello.c.
```

Byte 00H contains 80H, indicating a THEADR record.

Bytes 01-02H contain 0009H, the length of the remainder of the record.

Bytes 03-0AH contain the T-module name. Byte 03H contains 07H, the length of the name, and bytes 04H through 0AH contain the name itself (HELLO.C).

Byte 0BH contains the Checksum field, 0CBH.

82H LHEADR—Library Module 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 an internal organization different from that of an object module).

History

This record type was defined in the original Intel 8086 specification with the same format but with a different purpose, so its use for libraries should be considered a Microsoft extension.

Record Format

1	2	1	<-------String Length------>	1
82	Record Length	String Length	Name String	Checksum

Note: *The THEADR, and LHEADR records are handled identically. See Appendix 2 for a complete description of these library file format.*

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 accessing the object module, or even binary-encoded identification data. An object module can contain any number of COMENT records.

History

Before the VENDEXT record was added for TIS, the COMENT record was the primary way of extending the OMF. These extensions were added or changed for 32-bit linkers and continue to be supported in this standard. The comment classes that have been added or changed are 9D, A0, A1, A2, A4, AA, B0, and B1.

Comment class A2 was added for Microsoft C version 5.0. Histories for comment classes A0, A3, A4, A6, A7, and A8 are given later in this section.

68000 and big-endian comments were added for Microsoft C version 7.0.

Record Format

The comment records are actually a group of items, classified by comment class.

1	2	1	1	<---------Record Length Minus 3-------->	1
88	Record Length	Comment Type	Comment Class	Commentary Byte String (optional)	Checksum

Comment Type

The Comment Type field is bit significant; its layout is

<----------1 byte ---------->							
NP	NL	0	0	0	0	0	0

where

NP (no purge bit) is set if the comment is to be preserved by utility programs that manipulate object modules. This bit can protect an important comment, such as a copyright message, from deletion.

NL (no list bit) is set if the comment is not to be displayed by utility programs that list the contents of object modules. This bit can hide a comment.

The remaining bits are unused and should be set to 0.

Comment Class and Commentary Byte String

The Comment Class field is an 8-bit numeric field that 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 that have been defined (including obsolete values) are the following:

0	**Translator**	Translator; it may name the source language or translator. We recommend that the translator name and version, plus the 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.
2 – 9B	**Intel reserved**	These values are reserved for Intel products. Values from 9C through FF are ignored by Intel products.
81	**Library specifier—obsolete**	Replaced by comment class 9F; contents are identical to 9F.
9C	**MS-DOS version—obsolete**	The commentary byte string field is a 2 byte string that specifies the MS-DOS version number. This comment class is not supported by Microsoft LINK.
9D	**Memory model**	This information is currently generated by the Microsoft C compiler for use by the XENIX linker; it is ignored by the MS-DOS and OS/2 versions of Microsoft LINK. The byte string consists of from one to three ASCII characters and indicates the following:

0, 1, 2, or 3	8086, 80186, 80286, or 80386 instructions generated, respectively
O	Optimization performed on code
s, m, c, l, or h	Small, medium, compact, large, or huge model
A, B, C, D	68000, 68010, 68020, or 68030 instructions generated, respectively

9E	**DOSSEG**	Sets Microsoft LINK's 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 MS-DOS, OS/2, and accompanying language products.
9F	**Default library search name**	The byte string contains a library filename (without a lead count byte and without an extension), which is searched in order to resolve external references within the object module. The default library search can be overridden with LINK's /NODEFAULTLIBRARYSEARCH switch.

A0	**OMF extensions**		This class consists of a set of records, identified by subtype (first byte of commentary string). Values supported by some linkers are:
	01	**IMPDEF**	Import definition record. See the IMPDEF section for a complete description.
	02	**EXPDEF**	Export definition record. See the EXPDEF section for a complete description.
	03	**INCDEF**	Incremental compilation record. See the INCDEF section for a complete description.
	04	**Protected memory library**	32-bit linkers only; relevant only to 32-bit dynamic-link libraries (DLLs). This comment record is inserted in an object module by the compiler when it encounters the _loadds construct in the source code for a 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 data is protected from corruption. The _loadds keyword tells the compiler to emit modified function prolog code, which loads the DS segment register. (Normal functions don't need this.) When the flag is set in the .EXE header, the loader loads the selector of a protected memory area into DS while performing run-time fixups (relocations). All other DLLs 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**	Microsoft C++ linker directives record. See the LNKDIR section for a complete description.
	06	**Big-endian**	The target for this OMF is a big-endian machine, as opposed to little-endian. "Big-endian" describes an architecture for which the most significant byte of a multibyte value has the smallest address. "Little-endian" describes an architecture for which the least significant byte of a multibyte value has the smallest address.
	07	**PRECOMP**	When the Microsoft symbol and type information for this object file is emitted, the directory entry for $$TYPES is to be emitted as sstPreComp instead of sstTypes.
	08-FF		Reserved.

***Note**: The presence of any unrecognized subtype causes the linker to generate a fatal error.*

A1 **"New OMF" extension**

This comment class is now used solely to indicate the version of the symbolic debug information. If this comment class is not present, the version of the debug information produced is defined by the linker. For example, Microsoft LINK defaults to the oldest format of Microsoft symbol and type information.

This comment class was previously used to indicate that the obsolete method of communal representation through TYPDEF and EXTDEF pairs was not used and that COMDEF records were used instead. In current linkers, COMDEF records are always enabled, even without this comment record present.

The byte string is currently empty, but the planned future contents will be a version number (8-bit numeric field) followed by an ASCII character string indicating the symbol style. Values will be:

n,'C','V' **Microsoft symbol and type style**

n,'D','X' **AIX style**

n,'H','L' **IBM PM Debugger style**

A2 **Link Pass Separator**

This record conveys information to the linker about the organization of the file. The value of the first byte of the commentary string specifies the comment subtype. Currently, a single subtype is defined:

01 Indicates the start of records generated from Pass 2 of the linker. Additional bytes may follow, with their number determined by the Record Length field, but they will be ignored by the linker.

See the "Order of Records" section for information on which records must come before and after this comment.

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

Note*: This comment class may become obsolete with the advent of COMDAT records.*

A3 **LIBMOD** Library module comment record. Ignored by the linker; used only by the librarian. See the LIBMOD section for a complete description.

A4 **EXESTR** Executable string. See the EXESTR section for a complete description.

A6 **INCERR** Incremental compilation error. See the INCERR section for a complete description.

A7 **NOPAD** No segment padding. See the NOPAD section for a complete description.

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

A9 **LZEXT** Lazy Extern record. See the LZEXT section for a complete description.

DA **Comment** For random comment.

DB **Compiler** For pragma comment(compiler); version number.

DC **Date** For pragma comment(date stamp).

DD **Timestamp** For pragma comment(timestamp).

DF **User** For pragma comment(user). Sometimes used for copyright notices.

E9	**Dependency file (Borland)**	Used to show the include files that were used to build this .OBJ file.
FF	**Command line (Microsoft QuickC)**	Shows the compiler options chosen. May be obsolete. This record is also used by Phoenix Technology Ltd. for library comments.
C0H-FFH	**-**	Comment classes within this range that are not otherwise used are reserved for user-defined comment classes. In general, the VENDEXT record replaces the need for new user-defined comment classes.

Notes

Microsoft LIB ignores the Comment Type field.

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 cannot be the first or last record in an object module. (The first record must always be THEADR or LHEADR and the last must always be MODEND.)*

Examples

The following three examples are typical COMENT records taken from an object module generated by the Microsoft C Compiler.

This first example is a language-translator comment:

```
      0    1    2    3    4    5    6    7    8    9    A    B    C    D    E    F
0000  88   07   00   00   00   4D   53   20   43   6E                             .....MS Cn
```

Byte 00H contains 88H, indicating that this is a COMENT record.

Bytes 01-02H contain 0007H, the length of the remainder of the record.

Byte 03H (the Comment Type field) contains 00H. Bit 7 (no purge) is set to 0, indicating that this COMENT record may be purged from the object module by a utility program that manipulates object modules. Bit 6 (no list) is set to 0, indicating that this comment need not be excluded from any listing of the module's contents. The remaining bits are all 0.

Byte 04H (the Comment Class field) contains 00H, indicating that this COMENT record contains the name of the language translator that generated the object module.

Bytes 05H through 08H contain the name of the language translator, Microsoft C.

Byte 09H contains the Checksum field, 6EH.

The second example contains the name of an object library to be searched by default when Microsoft LINK processes the object module containing this COMENT record:

```
      0    1    2    3    4    5    6    7    8    9    A    B    C    D    E    F
0000  88   09   00   00   9F   53   4C   49   42   46   50   10                  .....SLIBFP
```

Byte 04H (the Comment Class field) contains 9FH, indicating that this record contains the name of a library for LINK to use to resolve external references.

Bytes 05-0AH contain the library name, SLIBFP. In this example, the name refers to the Microsoft C Compiler's floating-point function library, SLIBFP.LIB.

The last example indicates that Microsoft LINK should write the most recent format of Microsoft symbol and type information to the executable file.

```
      0    1    2    3    4    5    6    7    8    9    A    B    C    D    E    F
0000  88   06   00   00   A1   01   43   56   37                                  .....CV7
```

Byte 04H indicates the comment class, 0A1H.

Bytes 05-07H, which contain the comment string, are ignored by LINK.

88H IMPDEF—Import Definition Record (Comment Class A0, Subtype 01)

Description

This record describes the imported names for a module.

History

This comment class and subtype is a Microsoft extension added for OS/2 and Windows.

Record Format

One import symbol is described; the subrecord format is

1	1	<variable>	<variable>	2 or <variable>
01	Ordinal Flag	Internal Name	Module Name	Entry Ident

where:

01 Identifies the subtype as an IMPDEF. It determines the form of the Entry Ident field.

Ordinal Flag Is a byte; if 0, the import is identified by name. If nonzero, it is identified by ordinal. It determines the form of the Entry Ident field.

Internal Name Is 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.

Module Name Is in *count, char* string format and is the name of the module (a DLL) that supplies an export symbol matching this import.

Entry Ident Is an ordinal or the name used by the exporting module for the symbol, depending upon the Ordinal Flag.

If this field is an ordinal (Ordinal Flag is nonzero), it is a 16-bit word. If this is a name and the first byte of the name is 0, then the exported name is the same as the imported name (in the Internal Name field). Otherwise, it is the imported name in *count, char* string format (as exported by Module Name).

***Note**: IMPDEF records are created by an import library utility, IMPLIB, which builds an "import library" from a module definition file or DLL.*

88H EXPDEF—Export Definition Record (Comment Class A0, Subtype 02)

Description

This record describes the exported names for a module.

History

This comment class and subtype is a Microsoft extension added for Microsoft C version 5.1.

Record Format

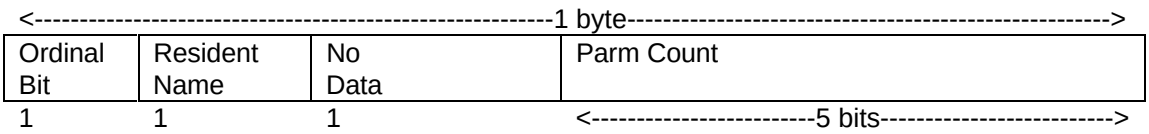
One exported entry point is described; the subrecord format is

1	1	<variable>	<variable>	2
02	Exported Flag	Exported Name	Internal Name	Export Ordinal
				<conditional>

where:

02 Identifies the subtype as an EXPDEF.

Exported Flag Is a bit-significant 8-bit field with the following format:



Ordinal Bit	Resident Name	No Data	Parm Count
1	1	1	<----------5 bits---------->

Ordinal Bit Is set if the item is exported by ordinal; in this case the Export Ordinal field is present.

Resident Name Is 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.

No Data Is set if the entry point does not use initialized data (either instanced or global).

Parm Count Is the number of parameter words. The Parm Count field is set to 0 for all but callgates to 16-bit segments.

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

Internal Name Is in *count, char* string format. If the name length is 0, the internal name is the same as the Exported Name field. Otherwise, it is the name by which the entry point is known within this module. This name will appear as a PUBDEF or LPUBDEF name.

Export Ordinal Is present if the Ordinal Bit field is set; it is a 16-bit numeric field whose value is the ordinal used (must be nonzero).

Note: *EXPDEFs are produced by many compilers when the keyword _export is used in a source file.*

88H INCDEF—Incremental Compilation Record (Comment Class A0, Subtype 03)

Description

This record is used for incremental compilation. Every FIXUPP and LINNUM record following 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 record per module.

History

This comment class subtype is a Microsoft extension added for Microsoft QuickC version 2.0.

Record Format

The subrecord format is:

1	2	2	<variable>
03	EXTDEF Delta	LINNUM Delta	Padding

The EXTDEF Delta and LINNUM Delta fields are signed.

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

***Note**: Negative deltas are allowed.*

88H LNKDIR—Microsoft C++ Directives Record (Comment Class A0, Subtype 05)

Description

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

History

This comment class and subtype is a Microsoft extension added for Microsoft C 7.0.

Record Format

The subrecord format is:

1	1	1	1
05	Bit Flags	Pseudocode Version	CodeView Version

Bit Flags Field

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 CodeView $PUBLICS	New .EXE

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

The second low-order bit indicates that the linker should not output the $PUBLICS subsection of the Microsoft symbol and type (CodeView) information.

The third low-order bit indicates the need to run the Microsoft Make Pseudocode Utility (MPC) over the object file to enable creation of an executable file.

Pseudocode Version Field

This is a one-byte field indicating the pseudocode interpreter version number.

CodeView Version Field

This is a one-byte field indicating the CodeView version number.

> ***Note:*** *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 librarian not by the linker. It gives the name of an object module within a library, allowing the librarian to preserve the source filename in the THEADR record and still identify the module names that make up the library. Since the module name is the base name of the .OBJ file that was built into the library, it may be completely different from the final library name.

History

This comment class and subtype is a Microsoft extension added for LIB version 3.07 in version 5.0 of its macro assembler (MASM).

Record Format

The subrecord format is:

1	<variable>
A3	Module Name

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

Notes

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

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

IBM LINK386 ignores LIBMOD comment records.

88H EXESTR—Executable String Record (Comment Class A4)

Description

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

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

History

This comment class and subtype is a Microsoft extension added for Microsoft C 5.1.

Record Format

The subrecord format is:

1	<variable>
A4	Arbitrary Text

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 Microsoft symbol and type information is present, the text will not be at the end of the file but somewhere before so as not to interfere with the Microsoft symbol and type information 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 a fatal error similar to "invalid object—error encountered during incremental compilation."

This behavior is useful when an incremental compilation fails and the user tries to link manually. The object module cannot be deleted, in order to preserve the base for the next incremental compilation.

History

This comment class and subtype is a Microsoft extension added for Microsoft QuickC 2.0.

Record Format

The subrecord format is:

1

A6	No Fields

88H NOPAD—No Segment Padding (Comment Class A7)

Description

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

History

This comment class and subtype is a Microsoft extension added for COBOL. It was added to Microsoft LINK to support MicroFocus COBOL version 1.2; it was added permanently in LINK version 5.11 to support Microsoft COBOL version 3.0.

Record Format

The subrecord format is:

1	1 or 2
A7	SEGDEF Index
	<----------repeated---------->

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

Note: *IBM LINK386 ignores NOPAD comment 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, the record associates another external name to use as the default resolution.

History

This comment class and subtype is a Microsoft extension added for Microsoft Basic version 7.0. There is no construct in Microsoft Basic that produces it, but the record type is manually inserted into Microsoft Basic library modules.

The first user-accessible construct to produce a weak extern was added for Microsoft MASM version 6.0.

See the following "Notes" section for details on how and why this record is used in Microsoft's Basic and MASM.

Record Format

The subrecord format is:

1	1 or 2	1 or 2
A8	Weak EXTDEF Index	Default Resolution EXTDEF Index
	<----------repeated---------->	

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

The Default Resolution EXTDEF Index field 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 a weak extern becomes strong, then it must be resolved with a matching PUBDEF, just like a regular EXTDEF. If a weak 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, many linkers 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.

Example

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

WKEXTs are best understood by explaining why they were added in the first place. The minimum BASIC run-time library in the past consisted of a large amount of code that 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 operations, 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.

Note:

The construct in Microsoft MASM 6.0 that produces a weak extern is

EXTERN var(con): byte

which makes "con" the default resolution for weak extern "var".

88H LZEXT—Lazy Extern Record (Comment Class A9)

Description

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

History

This comment class and subtype is a Microsoft extension added for Microsoft C 7.0, but was not implemented in the Microsoft C 7.0 linker.

Record Format

The subrecord format is:

1	1 or 2	1 or 2
A9	Lazy EXTDEF Index	Default Resolution EXTDEF Index

<----------repeated---------->

The Lazy EXTDEF Index field is the 1- or 2-byte index to the EXTDEF of the extern that is lazy.

The Default Resolution EXTDEF Index field 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 lazy extern is linked in*
- *If an EXTDEF for the lazy extern is found in another module (including libraries)*

If a lazy extern becomes strong, 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, many linkers will emit a warning.

Unlike weak externs, lazy externs do query libraries for resolution; if an extern is still lazy when libraries are searched, it stays lazy and gets the default resolution.

IBM LINK386 ignores LZEXT comment records.

8AH or 8BH MODEND—Module End Record

Description

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

History

Record type 8BH is was added for 32-bit linkers; it has a Target Displacement field of 32 bits rather than 16 bytes.

Record Format

1	2	1	1	1 or 2	1 or 2	2 or 4	1
8A or 8B	Record Length	Module Type	End Data	Frame Datum	Target Datum	Target Displacement	Checksum
			<----------Start Address subfield, conditional---------->				

where:

Module Type Field

The Module Type byte is bit significant; its layout is

MATTR Main Start	Segment Bit					
		0	0	0	0	X
<----------2 bits--------->						

where:

MATTR Is a 2-bit field.

Main Is set if the module is a main program module.

Start Is set if the module contains a start address; if this bit is set, the field starting with the End Data byte is present and specifies the start address.

Segment Bit Is reserved. Only 0 is supported by MS-DOS and OS/2.

X Is set if the Start Address subfield contains a relocatable address reference that the linker must fix up. (The original Intel 8086 specification allows this bit to be 0, to indicate that the start address is an absolute physical address that is stored as a 16-bit frame number and 16-bit offset, but this capability is not supported by certain linkers. This bit should always be set; however, the value will be ignored.

Start Address

The Start Address subfield is present only if the Start bit in the Module Type byte is set. Its format is identical to the Fix Data, Frame Datum, Target Datum, and Target Displacement fields in a FIXUP subrecord of a FIXUPP record. Bit 2 of the End Data field, which corresponds to the P bit in a Fix Data field, must be 0. The Target Displacement field (if present) is a 4-byte field if the record type is 8BH and a 2-byte field otherwise. This value provides the initial contents of CS:(E)IP.

If overlays are used, the start address must be given in the MODEND record of the root module.

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. If there are overlays, the linker needs to see the starting address on Pass 1 to define the symbol $$MAIN.

Example

Consider the MODEND record of a simple HELLO.ASM program:

```
      0    1    2    3    4    5    6    7    8    9    A    B    C    D    E    F
0000  8A   07   00   C1   00   01   01   00   00                                  AC.....
```

Byte 00H contains 8AH, indicating a MODEND record.

Bytes 01-02H contain 0007H, the length of the remainder of the record.

Byte 03H contains 0C1H (11000001B). Bit 7 is set to 1, indicating that this module is the main module of the program. Bit 6 is set to 1, indicating that a Start Address subfield is present. Bit 0 is set to 1, indicating that the address referenced in the Start Address subfield must be fixed up by the linker.

Byte 04H (End Data in the Start Address subfield) contains 00H. As in a FIXUPP record, bit 7 indicates that the frame for this fixup is specified explicitly, and bits 6 through 4 indicate that a SEGDEF index specifies the frame. Bit 3 indicates that the target reference is also specified explicitly, and bits 2 through 0 indicate that a SEGDEF index also specifies the target.

Byte 05H (Frame Datum in the Start Address subfield) contains 01H. This is a reference to the first SEGDEF record in the module, which in this example corresponds to the _TEXT segment. This reference tells the linker that the start address lies in the _TEXT segment of the module.

Byte 06H (Target Datum in the Start Address subfield) contains 01H. This also is a reference to the first SEGDEF record in the object module, which corresponds to the _TEXT segment. For example, Microsoft LINK uses the following Target Displacement field to determine where in the _TEXT segment the address lies.

Bytes 07-08H (Target Displacement in the Start Address subfield) contain 0000H. This is the offset (in bytes) of the start address.

Byte 09H contains the Checksum field, 0ACH.

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.

History

In the Intel specification and older linkers, the Type Index field was used as an index into TYPDEF records. This is no longer true; the field now encodes Microsoft symbol and type information (see Appendix 1 for details.) Many linkers ignore the old style TYPDEF.

Record Format

1	2	1	<String Length>	1 or 2	1
8C	Record Length	String Length	External Name String	Type Index	Checksum

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 (FIXUPP records); the name may not be null. Indexes start from 1.

String Length is a 1-byte field containing the length of the name field that follows it.

The Type Index field is encoded as an index field and contains debug information.

Notes

For Microsoft compilers, all referenced functions of global scope and all referenced variables explicitly declared "extern" will generate an EXTDEF record.

Many linkers impose a limit of 1023 external names and restrict the name length to a value between 1 and 7FH.

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 looks for a matching name in the following sequence:

1. *Searches PUBDEF and COMDEF records.*
2. *If linking a segmented executable, searches imported names (IMPDEF).*
3. *If linking a segmented executable and not a DLL, searches for an exported name (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, the default resolution is used. If the reference is to a strong extern, it's an undefined external, and the linker generates an error.*

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

Example

Consider this EXTDEF record generated by the Microsoft C Compiler:

```
       0  1  2  3   4   5  6  7   8  9  A  B  C   D   E  F
0000   8C 25 00 0A  5F  5F 61 63  72 74 75 73 65  64  00 05  .%.__acrtused..
0010   5F 6D 61 69  6E  00 05 5F  70 75 74 73 00  08  5F 5F  _main.._puts.__
0020   63 68 6B 73  74  6B 00 A5                            chkstk..
```

Byte 00H contains 8CH, indicating that this is an EXTDEF record.

Bytes 01-02H contain 0025H, the length of the remainder of the record.

Bytes 03-26H contain a list of external references. The first reference starts in byte 03H, which contains 0AH, the length of the name __acrtused. The name itself follows in bytes 04-0DH. Byte 0EH contains 00H, which indicates that the symbol's type is not defined by any TYPDEF record in this object module. Bytes 0F-26H contain similar references to the external symbols _main, _puts, and __chkstk.

Byte 27H contains the Checksum field, 0A5H.

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.

History

Record type 91H was added for 32-bit linkers; it has a Public Offset field of 32 bits rather than 16 bits.

Record Format

1	2	1 or 2	1 or 2	2	1	<String Length>	2 or 4	1 or 2	1
90 or 91	Record Length	Base Group Index	Base Segment Index	Base Frame	String Length	Public Name String	Public Offset	Type Index	Checksum
				<conditional>	<----------repeated---------->				

Base Group, Base Segment, and Base Frame Fields

The Base Group and Base Segment fields contain indexes specifying previously defined SEGDEF and GRPDEF records. The group index may be 0, meaning that no group is associated with this PUBDEF record.

The Base Frame field is present only if the Base Segment field is 0, but the contents of the Base Frame field are ignored.

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

According to the Intel 8086 specification, if both the segment and group indexes are 0, the Base Frame field contains a 16-bit paragraph (when viewed as a linear address); this may be used to define public symbols that are absolute. Absolute addressing is not fully supported by some linkers—it can be used for read-only access to absolute memory locations; however, writing to absolute memory locations may not work in some linkers.

Public Name String, Public Offset, and Type Index Fields

The Public Name String field is in *count, char* format and cannot be null. The maximum length of a Public Name is 255 bytes.

The Public Offset field 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 group, segment, or frame specified in the Base Group, Base Segment, or Base Frame fields.

The Type Index field is encoded in index format and contains either debug information or 0. The use of this field is determined by the OMF producer and typically provides type information for the referenced symbol.

Notes

All defined functions and initialized global variables generate PUBDEF records in most compilers. No PUBDEF record will be generated, however, for instantiated inline functions in C++.

Any PUBDEF records in an object module must appear after the GRPDEF and SEGDEF records to which they refer. Because PUBDEF records are not themselves referenced by any other type of object record, they are generally placed near the end of an object module.

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

Examples

The following two examples show PUBDEF records created by Microsoft's macro assembler, MASM.

The first example is the record for the statement:

```
PUBLIC  GAMMA
```

The PUBDEF record is:

```
       0   1   2   3   4   5   6   7   8   9   A   B   C   D   E   F
0000  90   0C  00  00  01  05  47  41  4D  4D  41  02  00  00  F9      .....GAMMA.....
```

Byte 00H contains 90H, indicating a PUBDEF record.

Bytes 01-02H contain 000CH, the length of the remainder of the record.

Bytes 03-04H represent the Base Group, Base Segment, and Base Frame fields. Byte 03H (the group index) contains 0, indicating that no group is associated with the name in this PUBDEF record. Byte 04H (the segment index) contains 1, a reference to the first SEGDEF record in the object module. This is the segment to which the name in this PUBDEF record refers.

Bytes 05-0AH represent the Public Name String field. Byte 05H contains 05H (the length of the name), and bytes 06-0AH contain the name itself, GAMMA.

Bytes 0B-0CH contain 0002H, the Public Offset field. The name GAMMA thus refers to the location that is offset two bytes from the beginning of the segment referenced by the Base Group, Base Segment, and Base Frame fields.

Byte 0DH is the Type Index field. The value of the Type Index field is 0, indicating that no data type is associated with the name GAMMA.

Byte 0EH contains the Checksum field, 0F9H.

The next example is the PUBDEF record for the following absolute symbol declaration:

```
            PUBLIC    ALPHA
ALPHA       EQU       1234h
```

The PUBDEF record is:

```
      0    1    2    3    4    5    6    7    8    9    A    B    C    D    E    F
0000  90   0E   00   00   00   00   00   05   41   4C   50   48   41   34   12   00  ...ALPHA4...
0010  B1
```

Bytes 03-06H (the Base Group, Base Segment, and Base Frame fields) contain a group index of 0 (byte 03H) and a segment index of 0 (byte 04H). Because both the group index and segment index are 0, a frame number also appears in the Base Group, Base Segment, and Base Frame fields. In this instance, the frame number (bytes 05-06H) also happens to be 0.

Bytes 07-0CH (the Public Name String field) contain the name ALPHA, preceded by its length.

Bytes 0D-0EH (the Public Offset field) contain 1234H. This is the value associated with the symbol ALPHA in the assembler EQU directive. If ALPHA is declared in another object module with the declaration

```
EXTRN   ALPHA:ABS
```

any references to ALPHA in that object module are fixed up as absolute references to offset 1234H in frame 0. In other words, ALPHA would have the value 1234H.

Byte 0FH (the Type Index field) contains 0.

94H or 95H LINNUM—Line Numbers Record

Description

The LINNUM record relates line numbers in source code to addresses in object code.

History

Record type 95H is added for 32-bit linkers; allowing for 32-bit debugger style-specific information.

Note: *For instantiated inline functions in Microsoft C 7.0, line numbers are output in LINSYM records with a reference to the function name instead of the segment.*

Record Format

1	2	1 or 2	1 or 2	<variable>	1
94 or 95	Record Length	Base Group	Base Segment	Debugger Style-specific Information	Checksum
				<---------repeated-------->	

Base Group and Base Segment Fields

The Base Group and Base Segment fields contain indexes specifying previously defined GRPDEF and SEGDEF records.

Notes

The debugger style-specific information is indicated by comment class A1.

Although the complete Intel 8086 specification allows the Base Group and Base Segment fields to refer to a group or to an absolute segment as well as to a relocatable segment, some linkers commonly restrict references in this field to relocatable segments.

The following discussion uses the Microsoft debugger style-specific information. The debugger style-specific information field in the LINNUM record is composed as follows:

2	2 or 4
Line Number	Line Number Offset
<----------repeated---------->	

For Microsoft LINK LINNUM records, the Line Number field contains a 16-bit quantity, in the range 0 through 7FFF and is, as its name indicates, a line number in the source code. The Line Number Offset field contains a 2-byte or 4-byte quantity that gives the translated code or data's start byte in the program segment defined by the SEGDEF index (4 bytes if the record type is 95H; 2 bytes for type 94H).

The Line Number and Line Number Offset fields can be repeated, so a single LINNUM record can specify multiple line numbers in the same segment.

Line Number 0 has a special meaning: it is used for the offset of the first byte after the end of the function. This is done so that the length of the last line (in bytes) can be determined.

The source file corresponding to a line number group is determined from the THEADR record.

Any LINNUM records in an object module must appear after the SEGDEF records to which they refer. Because LINNUM records are not themselves referenced by any other type of object record, they are generally placed near the end of an object module.

Also see the INCDEF record which is used to maintain line numbers after incremental compilation.

Example

The following LINNUM record was generated by the Microsoft C Compiler:

```
      0    1    2    3    4    5    6    7    8    9    A    B    C    D    E    F
0000  94   0F   00   00   01   02   00   00   00   03   00   08   00   04   00   0F   ..........
0010  00   3C                                                                        ..
```

Byte 00H contains 94H, indicating that this is a LINNUM record.

Bytes 01-02H contain 000FH, the length of the remainder of the record.

Bytes 03-04H represent the Base Group and Base Segment fields. Byte 03H (the Base Group field) contains 00H, as it must. Byte 04H (the Base Segment field) contains 01H, indicating that the line numbers in this LINNUM record refer to code in the segment defined in the first SEGDEF record in this object module.

Bytes 05-06H (the Line Number field) contain 0002H, and bytes 07-08H (the Line Number Offset field) contain 0000H. Together, they indicate that source-code line number 0002 corresponds to offset 0000H in the segment indicated in the Base Group and Base Segment fields.

Similarly, the two pairs of Line Number and Line Number Offset fields in bytes 09-10H specify that line number 0003 corresponds to offset 0008H and that line number 0004 corresponds to offset 000FH.

Byte 11H contains the Checksum field, 3CH.

96H LNAMES—List of Names Record

Description

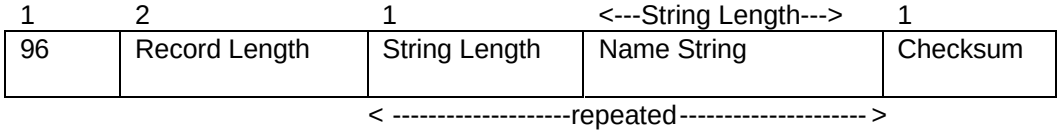
The LNAMES record is a list of names that can be referenced by subsequent SEGDEF and GRPDEF records in the object module.

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, group, overlay, and selector names.

History

This record has not changed since the original Intel 8086 OMF specification.

Record Format



Each name appears in *count, char* format, and a null name is valid. The character set is ASCII. Names can be up to 255 characters long.

Notes

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.

Previous versions limited the name string length to 254 characters.

Example

The following LNAMES record contains the segment and class names specified in all three of the following full-segment definitions:

```
_TEXT     SEGMENT byte public 'CODE'
_DATA     SEGMENT word public 'DATA'
_STACK    SEGMENT para public 'STACK'
```

The LNAMES record is:

```
      0   1   2   3   4   5   6   7   8   9   A   B   C   D   E   F
0000  96  25  00  00  04  43  4F  44  45  04  44  41  54  41  05  53  .%...CODE.DATA.S.
0010  54  41  43  4B  05  5F  44  41  54  41  06  5F  53  54  41  43  TACK._DATA._STAC
0020  4B  05  5F  54  45  58  54  8B                                  K._TEXT.
```

Byte 00H contains 96H, indicating that this is an LNAMES record.

Bytes 01-02H contain 0025H, the length of the remainder of the record.

Byte 03H contains 00H, a zero-length name.

Byte 04H contains 04H, the length of the class name CODE, which is found in bytes 05-08H. Bytes 09-26H contain the class names DATA and STACK and the segment names _DATA, _STACK, and _TEXT, each preceded by 1 byte that gives its length.

Byte 27H contains the Checksum field, 8BH.

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, and the way the segment can be combined with other logical segments at bind, link, or load time.

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

History

Record type 99H was added for 32-bit linkers: the Segment Length field is 32 bits rather than 16 bits. There is one newly implemented alignment type (page alignment), the B bit flag of the ACBP byte indicates a segment of 4 GB, and the P bit flag of the ACBP byte is the Use16/Use32 flag.

Starting with version 2.4, Microsoft LINK ignores the Overlay Name Index field. In versions 2.4 and later, command-line parameters to Microsoft LINK, rather than information contained in object modules, determine the creation of run-time overlays.

The length does not include COMDAT records. If selected, their size is added.

Record Format

1	2	<variable>	2 or 4	1 or 2	1 or 2	1 or 2	1
98 or 99	Record Length	Segment Attributes	Segment Length	Segment Name Index	Class Name Index	Overlay Name Index	Checksum

Segment Attributes Field

The Segment Attributes field is a variable-length field; its layout is:

<---3 bits--->	<---3 bits--->	<---1 bit--->	<---1 bit--->	<---2 bytes---->	<----1 byte---->
A	C	B	P	Frame Number <conditional>	Offset <conditional>

The fields have the following meanings:

A **Alignment**

A 3-bit field that specifies the alignment required when this program segment is placed within a logical segment. Its 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 a page boundary. (The original Intel 8086 specification defines a page to be 256 bytes. The IBM implementation of OMF uses a 4096-byte or 4K page size).

5 Relocatable, aligned on a double word (4-byte) boundary.

6 Not supported.

7 Not defined.

The new values for 32-bit linkers are A=4 and A=5. Double word alignment is expected to be useful as 32-bit memory paths become more prevalent. Page-align is useful for certain hardware-defined items (such as page tables) and error avoidance.

Note: *If A=0, the conditional Frame Number and Offset fields are present and indicate the starting address of the absolute segment. Microsoft LINK ignores the Offset field.*

Conflict*: The original Intel 8086 specification included additional segment-alignment values not supported by Microsoft; alignment 5 now conflicts with the following Microsoft LINK extensions:*

5 *"unnamed absolute portion of memory address space"*

6 *"load-time locatable (LTL), paragraph aligned if not part of any group"*

C Combination

This 3-bit field describes how the linker can combine the segment with other segments. Under MS-DOS, segments with the same name and class can be combined in two ways: they can be concatenated to form one logical segment, or they can be overlapped. In the latter case, they have either the same starting address or the same ending address, and they describe a common area in memory. Values for the C field are:

0 **Private.** Do not combine with any other program segment.

1 **Reserved.**

2 **Public.** Combine by appending at an offset that meets the alignment requirement.

3 **Reserved.**

4 Same as C=2 (public).

5 **Stack.** Combine as for C=2. This combine type forces byte alignment.

6 **Common.** Combine by overlay using maximum size.

7 Same as C=2 (public).

Conflict*: The original Intel 8086 specification lists C=1 as Common, not C=6.*

B **Big**

Used as the high-order bit of the Segment Length field. If this bit is set, the segment length value must be 0. If the record type is 98H and this bit is set, the segment is exactly 64K long. If the record type is 99H and this bit is set, the segment is exactly 2^{32} bytes or 4 GB long.

P This bit corresponds to the bit field for segment descriptors, known as the B bit for data segments and the D bit for code segments in Intel documentation.

If 0, then the 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 nonzero, then the segment may be larger than 64K (if data), and 32-bit addressing and operands are the default (if code). This is a Use32 segment.

***Note**: This is the only method for defining Use32 segments in the TIS OMF.*

Segment Length Field

The Segment Length field is a 2- or 4-byte numeric quantity and specifies the number of bytes in this program segment. For record type 98H, the length can be from 0 to 64K; if a segment is exactly 64K, the segment length should be 0, and the B field in the ACBP byte should be 1. For record type 99H, the length can be from 0 to 4 GB; if a segment is exactly 4 GB in size, the segment length should be 0 and the B field in the ACBP byte should be 1.

Segment Name Index, Class Name Index, Overlay Name Index Fields

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

The Segment Name Index field 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 a compiler.

The Class Name Index field identifies the segment with a class name (such as CODE, FAR_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 Name Index field identifies the segment with a run-time overlay. It is ignored by many linkers.

Notes

Many linkers impose a limit of 255 SEGDEF records per object module.

Certain name/class combinations are reserved for debug information and have special significance to the linker, such as $$TYPES and $$SYMBOLS. See Appendix 1 for more information.

***Conflicts:** The TIS-defined OMF has Use16/Use32 stored as the P bit of the ACBP field. The P bit does not specify the access for the segment. For Microsoft LINK the access is specified in the .DEF file.*

Examples

The following examples of Microsoft assembler SEGMENT directives show the resulting values for the A field in the corresponding SEGDEF object record:

```
aseg    SEGMENT at 400h                 ; A = 0
bseg    SEGMENT byte public 'CODE'      ; A = 1
cseg    SEGMENT para stack 'STACK'      ; A = 3
```

The following examples of assembler SEGMENT directives show the resulting values for the C field in the corresponding SEGDEF object record:

```
aseg    SEGMENT at 400H               ; C = 0
bseg    SEGMENT public 'DATA'         ; C = 2
cseg    SEGMENT stack 'STACK'         ; C = 5
dseg    SEGMENT common 'COMMON'       ; C = 6
```

In this first example, the segment is byte aligned:

```
      0    1    2    3    4    5    6    7    8    9    A    B    C    D    E    F
0000  98   07   00   28   11   00   07   02   01   1E                             ....(.....
```

Byte 00H contains 98H, indicating that this is a SEGDEF record.

Bytes 01-02H contain 0007H, the length of the remainder of the record.

Byte 03H contains 28H (00101000B), the ACBP byte. Bits 7-5 (the A field) contain 1 (001B), indicating that this segment is relocatable and byte aligned. Bits 4-2 (the C field) contain 2 (010B), which represents a public combine type. (When this object module is linked, this segment will be concatenated with all other segments with the same name.) Bit 1 (the B field) is 0, indicating that this segment is smaller than 64K. Bit 0 (the P field) is ignored and should be 0, as it is here.

Bytes 04-05H contain 0011H, the size of the segment in bytes.

Bytes 06-08H index the list of names defined in the module's LNAMES record. Byte 06H (the Segment Name Index field) contains 07H, so the name of this segment is the seventh name in the LNAMES record. Byte 07H (the Class Name Index field) contains 02H, so the segment's class name is the second name in the LNAMES record. Byte 08H (the Overlay Name Index field) contains 1, a reference to the first name in the LNAMES record. (This name is usually null, as MS-DOS ignores it anyway.)

Byte 09H contains the Checksum field, 1EH.

The second SEGDEF record declares a word-aligned segment. It differs only slightly from the first.

```
      0    1    2    3    4    5    6    7    8    9    A    B    C    D    E    F
0000  98   07   00   48   0F   00   05   03   01   01                             .. H......
```

Bits 7-5 (the A field) of byte 03H (the ACBP byte) contain 2 (010B), indicating that this segment is relocatable and word aligned.

Bytes 04-05H contain the size of the segment, 000FH.

Byte 06H (the Segment Name Index field) contains 05H, which refers to the fifth name in the previous LNAMES record.

Byte 07H (the Class Name Index field) contains 03H, a reference to the third name in the LNAMES record.

9AH GRPDEF—Group Definition Record

Description

This record causes the program segments identified by SEGDEF records to be collected together (grouped). For OS/2, the segments are combined into a logical segment that is to be addressed through a single selector. For MS-DOS, the segments are combined within the same 64K frame in the run-time memory map.

History

The special group name "FLAT" has been added for 32-bit linkers.

Record Format

1	2	1 or 2	1	1 or 2	1
9A	Record Length	Group Name Index	FF Index	Segment Definition	Checksum
			<--------- repeated --------->		

Group Name Field

The Group Name field contains an index into a previously defined LNAMES name and must be nonzero.

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

Group Components

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

The first byte of each group component is a type field for the remainder of the component. Certain linkers require a type value of FFH and always assume that the component contains a segment index value. See the "Notes" section below for other types defined by Intel.

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

Notes

Most linkers impose a limit of 31 GRPDEF records in a single object module and limit the total number of group definitions across all object modules to 31.

This record is frequently followed by a THREAD FIXUPP record.

A common grouping using the Group Definition Record would be to group the default data segments.

Most linkers do special handling of the pseudo-group name 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.

The additional group component types defined by the original Intel 8086 specification and the fields that follow them are:

FE	*External Index*
FD	*Segment Name Index, Class Name Index, Overlay Name Index*
FB	*LTL Data field, Maximum Group Length, Group Length*
FA	*Frame Number, Offset*

None of these types are supported by Microsoft LINK or IBM LINK386.

Example

The example of a GRPDEF record below corresponds to the following assembler directive:

```
tgroup  GROUP seg1,seg2,seg3
```

The GRPDEF record is:

```
       0    1    2    3    4    5    6    7    8    9    A    B    C    D    E    F
0000   9A   08   00   06   FF   01   FF   02   FF   03   55                        .....U
```

Byte 00H contains 9AH, indicating that this is a GRPDEF record.

Bytes 01-02H contain 0008H, the length of the remainder of the record.

Byte 03H contains 06H, the Group Name Index field. In this instance, the index number refers to the sixth name in the previous LNAMES record in the object module. That name is the name of the group of segments defined in the remainder of the record.

Bytes 04-05H contain the first of three group component descriptor fields. Byte 04H contains the required 0FFH, indicating that the subsequent field is a segment index. Byte 05H contains 01H, a segment index that refers to the first SEGDEF record in the object module. This SEGDEF record declared the first of three segments in the group.

Bytes 06-07H represent the second group component descriptor, this one referring to the second SEGDEF record in the object module.

Similarly, bytes 08-09H are a group component descriptor field that references the third SEGDEF record.

Byte 0AH contains the Checksum field, 55H.

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.

History

Record type 9DH was added for 32-bit linkers; it has a Target Displacement field of 32 bits rather than 16 bits, and the Location field of the Locat word has been extended to 4 bits (using the previously unused higher order S bit) to allow new LOCATION values of 9, 11, and 13.

Record Format

1	2	< ---------- from Record Length ---------->	1
9C or 9D	Record Length	THREAD subrecord or FIXUP subrecord	Checksum
		< ---------- repeated ---------->	

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 can be mixed within one object record.

Information that determines how to resolve a reference can be specified explicitly in a FIXUP subrecord, or it 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 subrecord can be referenced in several subsequent FIXUP subrecords, a FIXUPP object record that uses THREAD subrecords may be smaller than one in which THREAD subrecords 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 Subrecord

There are four FRAME threads and four 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 does not require that a previous LEDATA or LIDATA record occur.

The layout of the THREAD subrecord is as follows:

<---------- 1 byte ---------- >					<---------- 1 or 2 bytes ---------->
0	D	0	Method	Thred	Index
1	1	1	3	2 (bits)	<----------conditional---------->

where:

0 The high-order bit is 0 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 Fix Data field of 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 field.

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 Microsoft LINK or IBM LINK386).

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. Microsoft LINK and IBM LINK386 determine 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 any current linker.)

F4 The FRAME is determined by the segment index of the previous LEDATA or LIDATA record (that is, the segment in which the location is defined).

F5 The FRAME is determined by the TARGET's segment, group, or external index.

F6 Invalid.

Note: *The Index field is present for FRAME methods F0, F1, and F2 only.*

Thred A 2-bit field that determines the thread number (0 through 3, for the four threads of each kind).

Index A conditional field that contains an index value that refers to a previous SEGDEF, GRPDEF, or EXTDEF record. The field is present only 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 Subrecord

A FIXUP subrecord gives the how/what/why/where/who information required to resolve or relocate 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 subrecord. The FIXUP subrecord is as follows:

2	1	1 or 2	1 or 2	2 or 4
Locat	Fix Data	Frame Datum	Target Datum	Target Displacement
	<conditional>	<conditional>	<conditional>	

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 byte, rather than the high-order byte, as follows:

< ---------- low-order byte -------- >< --------high-order byte--------- >

1	M	Location	Data Record Offset
1	1	4	10 (bits)

where:

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

M Is the mode; M=1 for segment-relative fixups, and M=0 for self-relative fixups.

Location Is a 4-bit field that 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 High-order byte (high byte of 16-bit offset). Microsoft LINK and IBM LINK386 do not support this type.

5 16-bit loader-resolved offset, treated as Location=1.

***Conflict**: The PharLap implementation of OMF uses Location=5 to indicate a 32-bit offset, where IBM and Microsoft use Location=9.*

6 Not defined, reserved.

***Conflict**: The PharLap implementation of OMF uses Location=6 to indicate a 48-bit pointer (16-bit base:32-bit offset), where IBM and Microsoft use Location=11.*

7 Not defined, reserved.

8 Not defined, reserved.

9 32-bit offset.

10 Not defined, reserved.

11 48-bit pointer (16-bit base:32-bit offset).

12 Not defined, reserved.

13 32-bit loader-resolved offset, treated as Location=9 by the linker.

Data Record Offset Indicates the position of the LOCATION to be fixed up in the LEDATA or LIDATA record immediately preceding the FIXUPP record. This offset indicates either a byte in the Data Bytes field of an LEDATA record or a data byte in the Content field of a Data Block field in an LIDATA record.

The Fix Data bit layout is

F	Frame	T	P	Targt
1	3	1	1	2 (bits)

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 A 3-bit numeric field, 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 subrecord 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 Target Displacement field.

If P=0, the Target Displacement field is present. It is a 4-byte field if the record type is 9DH; it is a 2-byte field otherwise.

Targt A 2-bit numeric field, which gives the lower two bits of the TARGET method (if T=0) or gives the TARGET thread number (if T=1).

Frame Datum, Target Datum, and Target Displacement Fields

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

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

The Target Displacement field, a 16-bit or 32-bit field, is present only if the P bit in the Fix Data field is set 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 field 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 protected mode, the only legal segment register values are selectors; every segment and group of segments is mapped through some selector and addressed by an offset within the underlying memory defined by that selector.

Examples

For good examples of the usage of the FIXUP record, consult *The MS-DOS Encyclopedia*.

A0H or A1H LEDATA—Logical Enumerated Data Record

Description

This record provides contiguous binary data—executable code or program data—that 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 fixing up as determined by the presence of a subsequent FIXUPP record, but otherwise they require no expansion when mapped to memory at run time.

History

Record type A1H was added for 32-bit linkers; it has an Enumerated Data Offset field of 32 bits rather than 16 bits.

Record Format

1	2	1 or 2	2 or 4	<from Record Length>	1
A0 or A1	Record Length	Segment Index	Enumerated Data Offset	Data Bytes	Checksum

Segment Index Field

The Segment Index field 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 Field

The Enumerated Data Offset field is either a 2- or 4-byte field (depending on the record type) that 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 Field

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.

Notes

Record type A1H has the offset stored as a 32-bit value. Record type A0H encodes the offset value as a 16-bit numeric field (zero-extended if applied to a Use32 segment).

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

Code for functions is output in LEDATA records currently. The segment for code is usually named _TEXT (or module*_TEXT, depending on the memory model), unless #pragma alloc_text is used to specify a different code segment for the specified functions.*

For instantiated functions in Microsoft C++, code will simply be output in COMDAT records that refer to the function and identify the function's segment.

Data, usually generated by initialized variables (global or static), is output in LEDATA/LIDATA records referring to either a data segment or, possibly, a segment created for a based variable.

Example

The following LEDATA record contains a simple text string:

```
      0    1    2    3    4    5    6   7   8   9   A   B   C   D    E    F
0000  A0   13   00   02   00   00   48  65  6C  6C  6F  2C  20  77   6F   72  ...... Hello, wor
0010  6C   64   0D   0A   24   A8                                             ld..$.
```

Byte 00H contains 0A0H, which identifies this as an LEDATA record.

Bytes 01-02H contain 0013H, the length of the remainder of the record.

Byte 03H (the Segment Index field) contains 02H, a reference to the second SEGDEF record in the object module.

Bytes 04-05H (the Enumerated Data Offset field) contain 0000H. This is the offset, from the base of the segment indicated by the Segment Index field, at which the data in the Data Bytes field will be placed when the program is linked. Of course, this offset is subject to relocation by the linker because the segment declared in the specified SEGDEF record may be relocatable and may be combined with other segments declared in other object modules.

Bytes 06-14H (the Data Bytes field) contain the actual data.

Byte 15H contains the Checksum field, 0A8H.

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 can be modified by the linker if the LIDATA record is followed by a FIXUPP record, although this is not recommended.

History

Record type A3H was added for 32-bit linkers; it has Iterated Data Offset and Repeat Count fields of 32 bits rather than 16 bits.

Record Format

1	2	1 or 2	2 or 4	<from Record Length>	1
A2 or A3	Record Length	Segment Index	Iterated Data Offset	Data Block	Checksum
				<----------repeated-------->	

Segment Index and Iterated Data Offset Fields

The Segment Index and Iterated Data Offset fields (2 or 4 bytes) are the same as for an LEDATA record. The index must be nonzero. This indicates the segment and offset at which the data in this LIDATA record is to be placed when the program is loaded.

Data Block Field

The data blocks have the following form:

2 or 4	2	<from Block Count>
Repeat Count	Block Count	Content

Repeat Count Field

The Repeat Count field is a 16-bit or 32-bit value that determines the number of times the Content field is to be repeated. The Repeat Count field is 32 bits only if the record type is A3H.

Conflict: *The PharLap implementation of OMF uses a 16-bit repeat count even in 32-bit records.*

Block Count Field

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

0 Indicates that the Content field that follows is a 1-byte count value followed by count data bytes. The data bytes will be mapped to memory, repeated as many times as are specified in the Repeat Count field.

!= 0 Indicates that the Content field that follows is composed of one or more Data Block fields. The value in the Block Count field specifies the number of Data Block fields (recursive definition).

Notes

The Microsoft C Compiler generates LIDATA records for initialized data. For example:

```
static int a[100] = { 1, };
```

A FIXUPP record may occur after the LIDATA record; however, 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.

Example 1

```
02 00 02 00 03 00 00 00 02 40 41 02 00 00 00 02 50 51
```

is an iterated data block with 16-bit repeat counts that expands to:

```
40 41 40 41 40 41 50 51 50 51 40 41 40 41 40 41 50 51 50 51
```

Here, the outer data block has a repeat count of 2 and a block count of 2 (which means to repeat twice the result of expanding the two inner data blocks). The first inner data block has repeat count = 3, block count = 0. The content is 2 bytes of data (40 41); the repeat count expands the data to a string of 6 bytes. The second (and last) inner data block has a repeat count = 2, block count = 0, content 2 bytes of data (50 51). This expands to 4 bytes, which is concatenated with the 6 bytes from the first inner data block. The resulting 10 bytes are then expanded by 2 (the repeat count of the outer data block) to form the 20-byte sequence illustrated.

Example 2

This sample LIDATA record corresponds to the following assembler statement, which declares a 10-element array containing the strings ALPHA and BETA:

```
db      10 dup('ALPHA','BETA')
```

The LIDATA record is

```
        0   1   2   3   4   5   6   7   8   9   A   B   C   D   E   F
0000    A2  1B  00  01  00  00  0A  00  02  00  01  00  00  00  05  41  ...............A
0010    4C  50  48  41  01  00  00  00  04  42  45  54  41  A9          LPHA.....BETA.
```

Byte 00H contains 0A2H, identifying this as an LIDATA record.

Bytes 01-02H contain 1BH, the length of the remainder of the record.

Byte 03H (the Segment Index field) contains 01H, a reference to the first SEGDEF record in this object module, indicating that the data declared in this LIDATA record is to be placed into the segment described by the first SEGDEF record.

Bytes 04-05H (the Iterated Data Offset field) contain 0000H, so the data in this LIDATA record is to be located at offset 0000H in the segment designated by the segment.

Bytes 06-1CH represent an iterated data block:

- Bytes 06-07H contain the repeat count, 000AH, which indicates that the Content field of this iterated data block is to be repeated 10 times.

- Bytes 08-09H (the block count for this iterated data block) contain 0002H, which indicates that the Content field of this iterated data block (bytes 0A-1CH) contains two nested iterated data block fields (bytes 0A-13H and bytes 14-1CH).

- Bytes 0A-0BH contain 0001H, the repeat count for the first nested iterated data block. Bytes 0C-0DH contain 0000H, indicating that the Content field of this nested iterated data block contains data rather than more nested iterated data blocks. The Content field (bytes 0E-13H) contains the data; byte 0EH contains 05H, the number of subsequent data bytes; and bytes 0F-13H contain the actual data (the string ALPHA).

- Bytes 14-1CH represent the second nested iterated data block, which has a format similar to that of the block in bytes 0A-13H. This second nested iterated data block represents the 4-byte string BETA.

- Byte 1DH is the Checksum field, 0A9H.

B0H COMDEF—Communal Names Definition Record

Description

The COMDEF record is an extension to the basic set of 8086 object record types. It declares a list of one or more communal variables (uninitialized static data or data that may match initialized static data in another compilation unit).

The size of such a variable is 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 (noted below).

History

The COMDEF record was introduced by version 3.5 of Microsoft LINK.

This record is also used by Borland to support unique instantiatians of virtual tables, “out of inlines” and various thunks the C++ compiler generates. Borland’s documentation refers to this record as the VIRDEF record.

Record Format

1	2	1	<String Length>	1 or 2	1	<from Data Type>	1
B0	Record Length	String Length	Communal Name	Type Index	Data Type	Communal Length	Checksum
		<----------repeated---------->					

Communal Name Field

The name is in *count, char* format, and the 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 Field

This field encodes symbol information; it is parsed as an index field (1 or 2 bytes) and is not inspected by linkers.

Data Type and Communal Length Fields

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 Data Type field is one of the following hexadecimal values:

1 to 5FH Interpreted as a Borland segment index.

61H FAR data; the length is specified as the number of the elements followed by the element size in bytes

62H NEAR data; the length is specified as the number of bytes

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

Value Range	Number of Bytes	Allocation
0 through 128	1	This byte contains the value
0 to 64K-1	3	First byte is 81H, followed by a 16-bit word whose value is used
0 to 16 MB-1	4	First byte is 84H, followed by a 3-byte value
-2 GB-1 to 2 GB-1	5	First byte is 88H, followed by a 4-byte value

Groups of Communal Name, Type Index, Data 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 to which this module is bound or linked, certain linkers will give the error "symbol defined more than once."

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

The records are ordered by occurrence, together with the items named in EXTDEF and LEXTDEF records (for reference in FIXUP subrecords).

In older linkers, object modules that contain COMDEF records are required to also contain one COMENT record with comment class 0A1H, indicating that Microsoft extensions to the Intel object record specification are included in the object module. This COMENT record is no longer required; linkers always interpret COMDEF records.

Example

The following COMDEF record was generated by Microsoft C Compiler version 4.0 for these public variable declarations:

```
int     var;                    /* 2-byte integer */
char    var2[32768];            /* 32768-byte array */
char    far var3[10][2][20];    /* 400-byte array */
```

The COMDEF record is:

```
         0   1   2   3   4   5   6   7   8   9   A   B   C   D   E   F
0000  B0  20  00  04  5F  66  6F  6F  00  62  02  05  5F  66  6F  6F  .  .._var.b.._var
0010  32  00  62  81  00  80  05  5F  66  6F  6F  33  00  61  81  90  2.b...._var3.a..
0020  01  01  99                                                      ...
```

Byte 00H contains 0B0H, indicating that this is a COMDEF record.

Bytes 01-02H contain 0020H, the length of the remainder of the record.

Bytes 03-0AH, 0B-15H, and 16-21H represent three declarations for the communal variables var, var2, and var3. The Microsoft C compiler prepends an underscore to each of the names declared in the source code, so the symbols represented in this COMDEF record are _var, _var2, and _var3.

Byte 03H contains 04H, the length of the first Communal Name field in this record. Bytes 04-07H contain the name itself (_var). Byte 08H (the Type Index field) contains 00H, as required. Byte 09H (the Data Type field) contains 62H, indicating that this is a NEAR variable. Byte 0AH (the Communal Length field) contains 02H, the size of the variable in bytes.

Byte 0BH contains 05H, the length of the second Communal Name field. Bytes 0C-10H contain the name _var2. Byte 11H is the Type Index field, which again contains 00H, as required. Byte 12H (the Data Type field) contains 62H, indicating that _var2 is a NEAR variable.

Bytes 13-15H (the Communal Length field) contain the size in bytes of the variable. The first byte of the Communal Length field (byte 13H) is 81H, indicating that the size is represented in the subsequent two bytes of data—bytes 14-15H, which contain the value 8000H.

Bytes 16-1BH represent the Communal Name field for _var3, the third communal variable declared in this record. Byte 1CH (the Type Index field) again contains 00H as required. Byte 1DH (the Data Type field) contains 61H, indicating that this is a FAR variable. This means the Communal Length field is formatted as a Number of Elements field (bytes 1E-20H, which contain the value 0190H) and an Element Size field (byte 21H, which contains 01H). The total size of this communal variable is thus 190H times 1, or 400 bytes.

Byte 22H contains the Checksum field, 99H.

B2H or B3H BAKPAT—Backpatch Record

Description

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

History

Record type B2H is a Microsoft extension that was added for QuickC version 1.0. Record type B3H is the 32-bit equivalent: the Offset and Value fields are 32 bits rather than 16 bits.

Record Format

1	2	1 or 2	1	2 or 4	2 or 4	1
B2 or B3	Record Length	Segment Index	Location Type	Offset	Value	Checksum
				<----------repeated---------->		

Segment Index Field

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

Location Type Field

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

- **0** 8-bit low-order byte
- **1** 16-bit offset
- **2** 32-bit offset, record type B3H only (not supported yet)
- **9** 32-bit offset, per the IBM implementation of OMF

Offset and Value Fields

These fields are 32 bits for record type B3H, and 16 bits for B2H.

The Offset field specifies the LOCATION to be patched (as an offset into the SEGDEF record whose index is Segment Index).

The associated Value field is added to the LOCATION being patched (unsigned addition, ignoring overflow). The Value field is a fixed length (16 bits or 32 bits, 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 records do.

These records are buffered by the linker in Pass 2 until the end of the module, after the linker applies all other FIXUPP records. Most linkers then processes these records as fixups.

Example

To generate a self-relative address whose TARGET is a forward reference (JZ forwardlabel), the translator can insert the negative offset of the next instruction (-*) from the start of the SEGDEF record, followed by an additive backpatch (meaning that the backpatch is added to the original value and the sum replaces the original value) whose Value is the offset of the TARGET of the jump, which is done last.

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.

History

This record is an extension to the original set of 8086 object record types. It was added for Microsoft C 5.0. There is no semantic difference between the B4H and B5H types.

Record Format

1	2	1	<String Length>	1 or 2	1
B4 B5	Record Length	String Length	External Name String	Type Index	Checksum

<----------repeated---------->

Notes

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

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

Example

This record type is produced in Microsoft C from static functions, such as:

```
static int var() { }
```

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.

History

This record is an extension to the original set of 8086 object record types. It was added for Microsoft C 5.0. Record type B7H has been added for 32-bit linkers: the Local Offset field is 32 bits rather than 16 bits.

Record Format

1	2	1 or 2	1 or 2	2	1	<String Length>	2 or 4	1 or 2	1
B6 or B7	Record Length	Base Group	Base Segment	Base Frame	String Length	Local Name String	Local Offset	Type Index	Checksum
				<conditional>	<----------repeated---------->				

Note: *In Microsoft C, the static keyword on functions or initialized variables produces LPUBDEF records. Uninitialized static variables produce LCOMDEF records.*

B8H LCOMDEF—Local Communal Names Definition Record

Description

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

History

This record is an extension to the original set of 8086 object record types. It was added for Microsoft C 5.0.

Record Format

1	2	1	<String Length>	1 or 2	1	<from Data Type>	1
B8	Record Length	String Length	Communal Name	Type Index	Data Type	Communal Length	Checksum
		<----------repeated---------->					

Note: *In Microsoft C, uninitialized static variables produce an LCOMDEF record.*

BCH CEXTDEF—COMDAT External Names Definition Record

Description

This record serves the same purpose as the EXTDEF record described earlier. However, the symbol named is referred to through a Logical Name Index field. Such a Logical Name Index field is defined through an LNAMES or LLNAMES record.

History

The record is an extension to the original set of 8086 object record types. It was added for Microsoft C 7.0.

Record Format

1	2	1 or 2	1 or 2	1
BC	Record Length	Logical Name Index	Type Index	Checksum
		<----------repeated---------->		

Notes

A CEXTDEF can precede the COMDAT to which it will be resolved. In this case, the location of the COMDAT is not known at the time the CEXTDEF is seen.

This record is produced when a FIXUPP record refers to a COMDAT symbol.

C2H or C3H COMDAT—Initialized Communal Data Record

Description

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

History

The record is an extension to the original set of 8086 object record types. It was added for Microsoft C 7.0.

Record Format

1	2	1	1	1	2 or 4	1 or 2	1 or 2	1 or 2 [1] <var> [2]	1	1
C2 or C3	Record Length	Flags	Attributes	Align	Enumerated Data Offset	Type Index	Public Base	Public Name	Data	Checksum
									<repeated>	

Flags Field

This field contains the following defined bits:

01H Continuation bit. If clear, 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 Data field contains enumerated data; otherwise, the Data field contains iterated data, as in an LIDATA record.

04H Local bit (effectively an "LCOMDAT"). This is used in preference to LLNAMES.

08H Data in code segment. If the application is overlaid, this COMDAT must be forced into the root text. Also, do not apply FARCALLTRANSLATION to this COMDAT.

Note: This flag bit is not supported by IBM LINK386.

Attributes Field

This field contains two 4-bit fields: the Selection Criteria to be used and the Allocation Type, which is an ordinal specifying the type of allocation to be performed. Values are:

Selection Criteria (High-Order 4 Bits):

Bit	Selection Criteria	
00H	No match	Only one instance of this COMDAT allowed.
10H	Pick Any	Pick any instance of this COMDAT.
20H	Same Size	Pick any instance, but instances must have the same length or the linker will generate an error.

30H	Exact Match	Pick any instance, but checksums of the instances must match or the linker will generate an error. Fixups are ignored.
40H – F0H		Reserved.

Allocation Type (Low-Order 4 bits):

Bit	Allocation	
00H	Explicit	Allocate in the segment specified in the ensuing Base Group, Base Segment, and Base Frame fields.
01H	Far Code	Allocate as CODE16. The linker will create segments to contain all COMDATs of this type.
02H	Far Data	Allocate as DATA16. The linker will create segments to contain all COMDATs of this type.
03H	Code32	Allocate as CODE32. The linker will create segments to contain all COMDATs of this type.
04H	Data32	Allocate as DATA32. The linker will create segments to contain all COMDATs of this type.
05H - 0FH		Reserved.

Align Field

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

0 Use value from SEGDEF
1 Byte aligned
2 Word aligned
3 Paragraph (16 byte) aligned
4 Page aligned. (The original Intel specification uses 256-byte pages, the IBM OMF implementation uses 4096-byte pages.)
5 Double word (4 byte) aligned
6 Not defined
7 Not defined

Enumerated Data Offset Field

This field specifies an offset relative to the beginning location of the symbol specified in the Public Name Index field and defines the relative location of the first byte of the Data field. Successive data bytes in the Data field occupy higher locations of memory. This works very much like the Enumerated Data 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 Field

The Type Index field is encoded in index format; it contains either debug information or an old-style TYPDEF index. If this index is 0, there is no associated type data. Old-style TYPDEF indexes are ignored by most linkers. Linkers do not perform type checking.

Public Base Field

This field is conditional and is identical to the public base fields (Base Group, Base Segment, and Base Frame) stored in the PUBDEF record. This field is present only if the Allocation Type field specifies Explicit allocation.

Public Name Field

[1] Microsoft LINK recognizes this field as a regular logical name index (1 or 2 bytes).

[2] IBM LINK386 recognizes this field as a regular length-prefixed name.

Data Field

The Data 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

Record type C3H has an Enumerated Data Offset field of 32 bits.

While creating addressing frames, most linkers add the COMDAT data to the appropriate logical segments, adjusting their sizes. At that time, the offset at which the data that goes inside the logical segment is calculated. Next, the linker creates physical segments from adjusted logical segments and reports any 64K boundary overflows.

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

In Pass 2, only the selected occurrence of COMDAT data will be stored in virtual memory, fixed, and later written into the .EXE file.

COMDATs are allocated in the order of their appearance in the .OBJ files if no explicit ordering is given.

A COMDAT record cannot be continued across modules. A COMDAT record can be duplicated in a single module.

If any COMDAT record on a given symbol has the local bit set, all COMDAT records on that symbol have that bit set.

C4H or C5H LINSYM—Symbol Line Numbers Record

Description

This record will be used to output line numbers for functions specified through COMDAT records. Each LINSYM record is associated with a preceding COMDAT record.

History

This record is an extension to the original set of 8086 object record types. It was added for Microsoft C 7.0.

Record Format

1	2	1	1 or 2 [1] <var> [2]	2	2 or 4	1
C4 or C5	Record Length	Flags	Public Name	Line Number	Line Number Offset	Checksum
				<----------repeated---------->		

Flags Field

This field contains one defined bit:

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

Public Name Field

[1] Microsoft LINK recognizes this field as a regular logical name index indicating the name of the base of the LINSYM record.

[2] IBM LINK386 recognizes this field as a length-preceded name indicating the name of the base of the LINSYM record.

Line Number Field

An unsigned number in the range 0 to 65,535.

Line Number Offset Field

The offset relative to the base specified by the symbol name base. The size of this field depends on the record type.

Notes

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

This record is used to output line numbers for functions specified through COMDAT records. Often, the residing segment as well as the relative offsets of such functions is unknown at compile time, in that the linker is the final arbiter of such information. For such cases, most compilers will generate this record to specify the line number/offset pairs relative to a symbolic name.

This record will also be used to discard duplicate LINNUM information. If the linker encounters two or more LINSYM records with matching symbolic names (corresponding to multiple COMDAT records with the same name), the linker will keep the one that corresponds to the retained COMDAT.

LINSYM records must follow the COMDATs to which they refer. A LINSYM on a given symbol refers to the most recent COMDAT on the same symbol. LINSYMs inherit the "localness" of their COMDATs.

C6H ALIAS—Alias Definition Record

Description

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

History

The record is an extension to the original set of 8086 object record types for FORTRAN version 5.1 (Microsoft LINK version 5.13).

Record Format

1	2	<variable>	<variable>	1
C6	Record Length	Alias Name	Substitute Name	Checksum
		<----------repeated---------->		

Alias Name Field

A regular length-preceded name of the alias symbol.

Substitute Name Field

A regular length-preceded name of the substitute symbol.

Notes

This record consists 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 later, the PUBDEF overrides the alias. If another ALIAS record with a different substitute symbol is encountered, a warning is emitted by most linkers, and the second substitute symbol is used.

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 similar to a BAKPAT record, except that it refers to a COMDAT record by logical name index rather than an LIDATA or LEDATA record. NBKPAT records must immediately follow the COMDAT/FIXUPP block to which they refer.

History

This record is an extension to the original set of 8086 object record types. It was added for Microsoft C 7.0.

Record Format

1	2	1	1 or 2 [1] <var> [2]	2 or 4	2 or 4	1
C8 or C9	Record Length	Location Type	Public Name	Offset	Value	Checksum
				<-------repeated-------->		

Location Type Field

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

0 8-bit byte
1 16-bit word
2 32-bit double word, record type C9H only

Public Name Field

[1] Microsoft LINK recognizes this field as a regular logical name index of the COMDAT record to be back patched.

[2] IBM LINK386 recognizes this field as a length-preceded name of the COMDAT record to be back patched.

Offset and Value Fields

These fields are 32 bits for record type C8H, 16 bits for C9H.

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

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

CAH LLNAMES—Local Logical Names Definition Record

Description

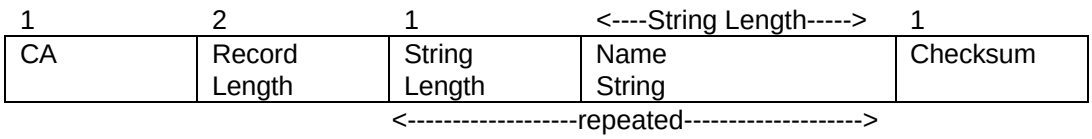
The LLNAMES record is a list of local names that can be referenced by subsequent SEGDEF and GRPDEF records in the object module.

The names are ordered by their occurrence, with the names in LNAMES records and referenced by index from subsequent records. More than one LNAMES and LLNAMES record may appear. The names themselves are used as segment, class, group, overlay, COMDAT, and selector names.

History

This record is an extension to the original set of 8086 object record types. It was added for Microsoft C 7.0.

Record Format



Each name appears in *count, char* format, and a null name is valid. The character set is ASCII. Names can be up to 255 characters long.

Notes

Any LLNAMES records in an object module must appear before the records that refer to them.

Previous versions limited the name string length to 254 characters.

CCH VERNUM - OMF Version Number Record

Description

The VERNUM record contains the version number of the object format generated. The version number is used to identify what version of the TIS-sponsored OMF was generated.

History

This is a new record that was approved by the Tool Interface Standards (TIS) Committee, an open industry standards body.

Record Format

1	2	1	<----String Length----->	1
CC	Record Length	Version Length	Version String	Checksum

The version string consists of 3 numbers separated by periods (.) as follows:

<TIS Version Base>.<Vendor Number>.<Version>

The TIS Version Base is the base version of the OMF being used. This number is provided by the TIS Committee. The Vendor Number is assigned by TIS to allow extensions specific to a vendor. Finally, the Version is the Vendor-specific version. A Vendor Number or Version of zero (0) is reserved for TIS. For example, a version string of 1.0.0 indicates a TIS compliant version of the OMF without vendor additions.

CEH VENDEXT - Vendor-specific OMF Extension Record

Description

The VENDEXT record allows vendor-specific extensions to the OMF. All vendor-specific extensions use this record.

History

This is a new record that was approved by the Tool Interface Standards (TIS) Committee, an open industry standards body.

Record Format

1	2	2	<from record length>	1
CE	Record Length	Vendor Number	Extension Bytes	Checksum

The Vendor Number is assigned by the TIS Committee. Zero (0) is reserved. The Extension Bytes provide OMF extension information.

Appendix 1: Microsoft Symbol and Type Extensions

Microsoft symbol and type information is stored on a per-module basis in specially-named logical segments. These segments are defined in the usual way (SEGDEF records), but the linker handles them specially, and they do not end up as segments in the .EXE file. These segment names are reserved:

Segment Name	Class Name	Combine Type
$$TYPES	DEBTYP	Private
$$SYMBOLS	DEBSYM	Private

The segment $$IMPORT should also be considered a reserved name, although it is not used anymore. This segment was not part of any object files but was emitted by the linker to get the loader to automatically do fixups for Microsoft symbol and type information. The linker emitted a standard set of imports, not just ones referenced by the program being linked. Use of this segment may be revisited in the future.

Microsoft symbol and type information-specific data is stored in LEDATA records for the $$TYPES and $$SYMBOLS segments, in various proprietary formats. The $$TYPES segment contains information on user-defined variable types; $$SYMBOLS contains information about nonpublic symbols: stack, local, procedure, block start, constant, and register symbols and code labels.

For instantiated functions in Microsoft C 7.0, symbol information for Microsoft symbol and type information will be output in COMDAT records that refer to segment $$SYMBOLS and have decorated names based on the function names. Type information will still go into the $$TYPES segment in LEDATA records.

All OMF records that specify a Type Index field, including EXTDEF, PUBDEF, and COMDEF records, use Microsoft symbol and type information values. Because many types are common, Type Index values in the range 0 through 511 (1FFH) are reserved for a set of predefined primitive types. Indexes in the range 512 through 32767 (200H-7FFFH) index into the set of type definitions in the module's $$TYPES segment, offset by 512. Thus 512 is the first new type, 513 the second, and so on.

Appendix 2: Library File Format

The first record in the library is a header that looks like any other object module format record.

Note: *Libraries under MS-DOS are always multiples of 512-byte blocks.*

Library Header Record (*n* bytes)

1	2	4	2	1	<n - 10>
Type (F0H)	Record Length (Page Size Minus 3)	Dictionary Offset	Dictionary Size in Blocks	Flags	Padding

The first byte of the record identifies the record's type, and the next two bytes specify the number of bytes remaining in the record. Note that this word field is byte-swapped (that is, the low-order byte precedes the high-order byte). The record type for this library header is F0H (240 decimal).

The Record Length field specifies the page size within the library. Modules in a library always start at the beginning of a page. Page size is determined by adding three to the value in the Record Length field (thus the header record always occupies exactly one page). Legal values for the page size are given by 2 to the *n*, where *n* is greater than or equal to 4 and less than or equal to 15.

The four bytes immediately following the Record Length field are a byte-swapped long integer specifying the byte offset within the library of the first byte of the first block of the dictionary.

The next two bytes are a byte-swapped word field that specifies the number of blocks in the dictionary.

Note: the MS-DOS Library Manager cannot create a library whose dictionary would require more than 251 512-byte pages.

The next byte contains flags describing the library. The current flag definition is:

0x01 = case sensitive (applies to both regular and extended dictionaries)

All other values are reserved for future use and should be 0.

The remaining bytes in the library header record are not significant. This record deviates from the typical OMF record in that the last byte is not used as a checksum on the rest of the record.

Immediately following the header is the first object module in the library. It, in turn, is succeeded by all other object modules in the library. Each module is in object module format (that is, it starts with a LHEADR record and ends with a MODEND record). Individual modules are aligned so as to begin at the beginning of a new page. If, as is commonly the case, a module does not occupy a number of bytes that is exactly a multiple of the page size, then its last block will be padded with as many null bytes as are required to fill it.

Following the last object module in the library is a record that serves as a marker between the object modules and the dictionary. It also resembles an OMF record.

Library End Record (marks end of objects and beginning of dictionary)

1	2	<n>
Type (F1H)	Record Length	Padding

The record's Type field contains F1H (241 decimal), and its Record Length field is set so that the dictionary begins on a 512-byte boundary. The record contains no further useful information; the remaining bytes are insignificant. As with the library header, the last byte is not a checksum.

Dictionary

The remaining blocks in the library compose the dictionary. The number of blocks in the dictionary is given in the library header. The dictionary provides rapid searching for a name using a two-level hashing scheme.

Due to the hashing algorithm, the number of dictionary blocks must be a prime number, and within each block is a prime number of buckets. Whereas a librarian can choose the prime number less than 255 for the dictionary blocks, the number of buckets within a block is fixed at 37.

To search for a name within the blocks, two hashing indices and two hash deltas are computed. A block index and block delta controls how to go from one block to the other, and a bucket index and bucket delta controls how to search buckets within a block. Each bucket within a block corresponds to a single string.

A block is 512 bytes long and the first 37 bytes correspond to the 37 buckets. To find the string corresponding to a bucket, multiply the value stored in the byte by two and use that as an index into the block. At this location in the block lies an unsigned byte value for the string length, followed by the string characters (not 0-terminated), which in turn is followed by a two-byte little-endian-format module number in which the module in the library defining this string can be found. Thus, all strings start at even locations in the block.

Byte 38 in a block records the free space left for storing strings in the block, and is an index of the same format as the bucket indices; that is, multiply the bucket index by two to find the next available slot in the block. If byte 38 has the value 255, there is no space left.

Dictionary Record (length is the dictionary size in 512-byte blocks)

37	1	<variable>	2	<conditional>
HTAB	FFLAG	Name	Block Number	Align Byte
		< ---------- repeated ---------- >		

Entries consist of the following: the first byte is the length of the symbol to follow, the following bytes are the text of the symbol, and the last two bytes are a byte-swapped word field that specifies the page number (counting the library header as page 0) at which the module defining the symbol begins.

All entries may have at most one trailing null byte in order to align the next entry on a word boundary.

Module names are stored in the LHEADR record of each module.

Extended Dictionary

The extended dictionary is optional and indicates dependencies between modules in the library. Versions of LIB earlier than 3.09 do not create an extended dictionary. The extended dictionary is placed at the end of the library.

The dictionary is preceded by these values:

BYTE =0xF2 Extended Dictionary header
WORD length of extended dictionary in bytes excluding first three bytes

Start of extended dictionary:

WORD number of modules in library = N

Module table, indexed by module number, with N + 1 fixed-length entries:

WORD module page number
WORD offset from start of extended dictionary to list of required modules

Last entry is null.

Dictionary Hashing Algorithm

Pseudocode for creating a library and inserting names into a dictionary is listed below.

```
    typedef unsigned short hash_value;
    typedef struct {
       hash_value block_x, block_d, bucket_x, bucket_d;
       } hash;
    typedef unsigned char block[512];
    unsigned short nblocks = choose some prime number such that it is
       likely that all the names will fit in those blocks (the value must be
       greater than 1 and less than 255);
    const int nbuckets = 37;
    const int freespace = nbuckets+1;
MORE_BLOCKS: ;
    // Allocate storage for the dictionary:
    block *blocks = malloc(nblocks*sizeof(block));
    // Zero out each block.
    memset(blocks,0,nblocks*sizeof(block));
    // Initialize freespace pointers.
    for (int i = 0; i < nblocks; i++)
       blocks[i][freespace] = freespace/2;

    for N <- each name you want to insert in the library do {
       int length_of_string = strlen(N);       // # of characters.
       // Hash the name, producing the four values (see below
       // for hashing algorithm):
       hash h = compute_hash(N, nblocks);
       hash_value start_block = h.block_x, start_bucket = h.bucket_x;
       // Space required:
       //         1 for len byte; string text; 2 bytes for module number;
       //         1 possible byte for pad.
       int space_required=1+length_of_string+2;
       if (space_required % 2) space_required++;      // Make sure even.
    NEXT_BLOCK: ;
       // Obtain pointer to block:
       unsigned char *bp = blocks[h.block_x];
       boolean success = FALSE;
       do {
            if (bp[h.bucket_x] == 0) {
                  if (512-bp[freespace]*2 < space_required) break;
                 // Found space.
                  bp[h.bucket_x] = bp[freespace];
                 int store_at = 2*bp[h.bucket_x];
                 bp[store_at] = length_of_string;
                 bp[store_at+1..store_at+length_of_string] = string characters;
                 int mod_location = store_at+length_of_string;
                 // Put in the module page number, LSB format.
                 bp[mod_location] = module_page_number % 256;
                 bp[mod_location+1] = module_page_number / 256;
                 bp[freespace] += space_required/2;
                 // In case we are right at the end of the block,
                 // set block to full.
                 if (bp[freespace] == 0) bp[freespace] = 0xff;
                 success = TRUE;
```

```
                break;
                }
           h.bucket_x = (h.bucket_x+h.bucket_d) % nbuckets;
           } while (h.bucket_x != start_bucket);
      if (!success) {
           // If we got here, we found no bucket.  Go to the next block.
           h.block_x = (h.block_x + h.block_d) % nblocks;
           if (h.block_x == start_block) {
                // We got back to the start block; there is no space
                // anywhere.  So increase the number of blocks to the
                // next prime number and start all over with all names.
                do nblocks++; while (nblocks is not prime);
                free(blocks);
                goto MORE_BLOCKS;
                }
         // Whenever you can't fit a string in a block, you must mark
         // the block as full, even though there may be enough space
         // to handle a smaller string.  This is because the linker,
         // after failing to find a string in a block, will decide
         // the string is undefined if the block has any space left.
           bp[freespace] = 0xff;
           goto NEXT_BLOCK;
           }
       }
```

The order of applying the deltas to advance the hash indices is critical, due to the behavior of the linker. For example, it would not be correct to check a block to see if there is enough space to store a string before checking the block's buckets, because this is not the way the linker functions. The linker does not restart at bucket 0 when it moves to a new block. It resumes at the bucket last used in the previous block. Thus, the librarian must move through the buckets, even though there is not enough room for the string, so that the final bucket index is the same one the linker arrives at when it finishes searching the block.

The algorithm to compute the four hash indices is listed below.

```
hash compute_hash(const unsigned char*  name, int blocks) {
    int len = strlen(name);
    const unsigned char *pb = name, *pe = name+len;
    const int blank = 0x20;    // ASCII blank.
    hash_value
         // Left-to-right scan:
        block_x = len | blank, bucket_d = block_x,
        // Right-to-left scan:
        block_d = 0, bucket_x = 0;
    #define rotr(x,bits) ((x << 16-bits) | (x >>    bits))
    #define rotl(x,bits) ((x <<    bits) | (x >> 16-bits))
    while (1) {
        // blank -> convert to LC.
        unsigned short cback = *--pe | blank;
        bucket_x = rotr(bucket_x,2) ^ cback;
        block_d = rotl(block_d,2) ^ cback;
        if (--len == 0) break;
        unsigned short cfront = *pb++ | blank;
        block_x = rotl(block_x,2) ^ cfront;
        bucket_d = rotr(bucket_d,2) ^ cfront;
        }
    hash h;
    h.block_x = block_x % blocks;
    h.block_d = _max(block_d % blocks,1);
    h.bucket_x = bucket_x % nbuckets;
    h.bucket_d = _max(bucket_d % nbuckets,1);
    return h;
    }
```

Appendix 3: Obsolete Records and Obsolete Features of Existing Records

This appendix contains a complete list of records that have been defined in the past but are not part of the TIS OMF. These record types are followed by a descriptive paragraph from the original Intel 8086 specification. When linkers encounter these records, they are free to process them, ignore them, or generate an error.

Obsolete Records

6EH RHEADR R-Module Header Record

This record serves to identify a module that has been processed (output) by Microsoft LINK-86/LOCATE-86. It also specifies the module attributes and gives information on memory usage and need.

70H REGINT Register Initialization Record

This record provides information about the 8086 register/register-pairs: CS and IP, SS and SP, DS and ES. The purpose of this information is for a loader to set the necessary registers for initiation of execution.

72H REDATA Relocatable Enumerated Data Record

This record provides contiguous data from which a portion of an 8086 memory image may eventually be constructed. The data may be loaded directly by an 8086 loader, with perhaps some base fixups. The record may also be called a Load-Time Locatable (LTL) Enumerated Data Record.

74H RIDATA Relocatable Iterated Data Record

This record provides contiguous data from which a portion of an 8086 memory image may eventually be constructed. The data may be loaded directly by an 8086 loader, but data bytes within the record may require expansion. The record may also be called a Load-Time Locatable (LTL) Iterated Data Record.

76H OVLDEF Overlay Definition Record

This record provides the overlay's name, its location in the object file, and its attributes. A loader may use this record to locate the data records of the overlay in the object file.

78H ENDREC End Record

This record is used to denote the end of a set of records, such as a block or an overlay.

7AH BLKDEF Block Definition Record

This record provides information about blocks that were defined in the source program input to the translator that produced the module. A BLKDEF record will be generated for every procedure and for every block that contains variables. This information is used to aid debugging programs.

7CH BLKEND Block End Record

This record, together with the BLKDEF record, provides information about the scope of variables in the source program. Each BLKDEF record must be followed by a BLKEND record. The order of the BLKDEF, debug symbol records, and BLKEND records should reflect the order of declaration in the source module.

7EH **DEBSYM** **Debug Symbols Record**

This record provides information about all local symbols, including stack and based symbols. The purpose of this information is to aid debugging programs.

84H **PEDATA** **Physical Enumerated Data Record**

This record provides contiguous data, from which a portion of an 8086 memory image may be constructed. The data belongs to the "unnamed absolute segment" in that it has been assigned absolute 8086 memory addresses and has been divorced from all logical segment information.

86H **PIDATA** **Physical Iterated Data Record**

This record provides contiguous data, from which a portion of an 8086 memory image may be constructed. It allows initialization of data segments and provides a mechanism to reduce the size of object modules when there is repeated data to be used to initialize a memory image. The data belongs to the "unnamed absolute segment."

8EH **TYPDEF** This record contains details about the type of data represented by a name declared in a PUBDEF or EXTDEF record. For more details on this record, refer to its description later in this appendix.

92H **LOCSYM** **Local Symbols Record**

This record provides information about symbols that were used in the source program input to the translator that produced the module. This information is used to aid debugging programs. This record has a format identical to the PUBDEF record.

9EH **(none)** **Unnamed record**

This record number was the only even number not defined by the original Intel 8086 specification. Apparently it was never used.

A4H **LIBHED** **Library Header Record**

This record is the first record in a library file. It immediately precedes the modules (if any) in the library. Following the modules are three more records in the following order: LIBNAM, LIBLOC, and LIBDIC.

A6H **LIBNAM** **Library Module Names Record**

This record lists the names of all the modules in the library. The names are listed in the same sequence as the modules appear in the library.

A8H **LIBLOC** **Library Module Locations Record**

This record provides the relative location, within the library file, of the first byte of the first record (either a THEADR or LHEADR or RHEADR record) of each module in the library. The order of the locations corresponds to the order of the modules in the library.

AAH **LIBDIC** **Library Dictionary Record**

This record gives all the names of public symbols within the library. The public names are separated into groups; all names in the *n*th group are defined in the *n*th module of the library.

8EH TYPDEF—Type Definition Record

Description

The TYPDEF record contains details about the type of data represented by a name declared in a PUBDEF or an EXTDEF record. This information may be used by a linker to validate references to names, or it may be used by a debugger to display data according to type.

Although the original Intel 8086 specification allowed for many different type specifications, such as scalar, pointer, and mixed data structure, many linkers used TYPDEF records to declare only communal variables. Communal variables represent globally shared memory areas—for example, FORTRAN common blocks or uninitialized public variables in Microsoft C. This function is served by the COMDEF record.

The size of a communal variable is declared explicitly in the TYPDEF record. If a communal variable has different sizes in different object modules, the linker uses the largest declared size when it generates an executable module.

History

Starting with Microsoft LINK version 3.5, the COMDEF record should be used for declaration of communal variables. However, for compatibility, later versions of Microsoft LINK recognize TYPDEF records as well as COMDEF records.

Record Format

1	2	<variable>	1	<variable>	1
8E	Record Length	Name	0 (EN)	Leaf Descriptor	Checksum

The name field of a TYPDEF record is in *count, char* format and is always ignored. It is usually a 1-byte field containing a single 0 byte.

The Eight-Leaf Descriptor field in the original Intel 8086 specification was a variable-length (and possibly repeated) field that contained as many as eight "leaves" that could be used to describe mixed data structures. Microsoft uses a stripped-down version of the Eight-Leaf Descriptor, of which the first byte, the EN byte, is always set to 0.

The Leaf Descriptor field is a variable-length field that describes the type and size of a variable. The two possible variable types are NEAR and FAR.

If the field describes a NEAR variable (one that can be referenced as an offset within a default data segment), the format of the Leaf Descriptor field is:

1	1	<variable>
62H	Variable Type	Length in Bits

The 1-byte field containing 62H signifies a NEAR variable.
The Variable Type field is a 1-byte field that specifies the variable type:

77H Array
79H Structure
7BH Scalar

This field must contain one of the three values given above, but the specific value is ignored by most linkers.

The Length in Bits field is a variable-length field that indicates the size of the communal variable. Its format depends on the size it represents.

If the first byte of the size is 128 (80H) or less, then the size is that value. If the first byte of the size is 81H, then a 2-byte size follows. If the first byte of the size is 84H, then a 3-byte size follows. If the first byte of the size is 88H, then a 4-byte size follows.

If the Leaf Descriptor field describes a FAR variable (one that must be referenced with an explicit segment and offset), the format is:

1	1	<variable>	<variable>
61H	Variable Type (77H)	Number of Elements	Element Type Index

The 1-byte field containing 61H signifies a FAR variable.

The 1-byte variable type for a FAR communal variable is restricted to 77H (array). (As with the NEAR Variable Type field, the linker ignores this field, but it must have the value 77H.)

The Number of Elements field is a variable-length field that contains the number of elements in the array. It has the same format as the Length in Bits field in the Leaf Descriptor field for a NEAR variable.

The Element Type Index field is an index field that references a previous TYPDEF record. A value of 1 indicates the first TYPDEF record in the object module, a value of 2 indicates the second TYPDEF record, and so on. The TYPDEF record referenced must describe a NEAR variable. This way, the data type and size of the elements in the array can be determined.

***Note**: Microsoft LINK limits the number of TYPDEF records in an object module to 256.*

Examples

The following three examples of TYPDEF records were generated by Microsoft C Compiler version 3.0. (Later versions use COMDEF records.)

The first sample TYPDEF record corresponds to the public declaration:

```
int      var;     /* 16-bit integer */
```

The TYPDEF record is:

```
       0    1    2    3    4    5    6    7    8    9    A    B    C    D    E    F
0000   8E   06   00   00   00   62   7B   10   7F                                  .....b{..
```

Byte 00H contains 8EH, indicating that this is a TYPDEF record.

Bytes 01-02H contain 0006H, the length of the remainder of the record.

Byte 03H (the name field) contains 00H, a null name.

Bytes 04-07H represent the Eight-Leaf Descriptor field. The first byte of this field (byte 04H) contains 00H. The remaining bytes (bytes 05-07H) represent the Leaf Descriptor field:

- Byte 05H contains 62H, indicating that this TYPDEF record describes a NEAR variable.
- Byte 06H (the Variable Type field) contains 7BH, which describes this variable as scalar.
- Byte 07H (the Length in Bits field) contains 10H, the size of the variable in bits.

Byte 08H contains the Checksum field, 7FH.

The next example demonstrates how the variable size contained in the Length in Bits field of the Leaf Descriptor field is formatted:

```
char    var2[32768];   /* 32 KB array */
```

The TYPDEF record is:

```
      0   1   2   3   4   5   6   7   8   9   A   B   C   D   E   F
0000  8E  09  00  00  00  62  7B  84  00  00  04  04                  .....bc{.....
```

The Length in Bits field (bytes 07-0AH) starts with a byte containing 84H, which indicates that the actual size of the variable is represented as a 3-byte value (the following three bytes). Bytes 08-0AH contain the value 040000H, the size of the 32K array in bits.

This third Microsoft C statement, because it declares a FAR variable, causes two TYPDEF records to be generated:

```
char    far  var3[10][2][20];    /* 400-element FAR array*/
```

The two TYPDEF records are:

```
      0   1   2   3   4   5   6   7   8   9   A   B   C   D   E   F
0000  8E  06  00  00  62  7B  08  87  8E  09  00  00  00  00  61  77  ....bc{......aw
0010  81  90  01  01  7E                                              .....|
```

Bytes 00-08H contain the first TYPDEF record, which defines the data type of the elements of the array (NEAR, scalar, 8 bits in size).

Bytes 09-14H contain the second TYPDEF record. The Leaf Descriptor field of this record declares that the variable is FAR (byte 0EH contains 61H) and an array (byte 0FH, the variable type, contains 77H).

> ***Note**: Because this TYPDEF record describes a FAR variable, bytes 10-12H represent a Number of Elements field. The first byte of the field is 81H, indicating a 2-byte value, so the next two bytes (bytes 11-12H) contain the number of elements in the array, 0190H (400D).*

Byte 13H (the Element Type Index field) contains 01H, which is a reference to the first TYPDEF record in the object module—in this example, the one in bytes 00-08H.

PharLap Extensions to The SEGDEF Record (Obsolete Extension)

The following describes an obsolete extension to the SEGDEF record.

In the PharLap 32-bit OMF, there is an additional optional field that follows the Overlay Name Index field. The reserved bits should always be 0. The format of this field is

< ----------5 bits---------- >	<--1 bit-->	<---------2 bits ----- >
Reserved	U	AT

where **AT** is the access type for the segment and has the following possible values

0 Read only
1 Execute only
2 Execute/read
3 Read/write

and **U** is the Use16/Use32 bit for the segment and has the following possible values:

0 Use16
1 Use32

Conflict: The PharLap OMF uses a 16-bit Repeat Count field, even in 32-bit records.