8086

RELOCATABLE OBJECT MODULE

FORMATS

An Intel Technical Specification

Order Number: 121748-001

Intel Corporation, 3065 Bowers Avenue, Santa Clara, California 95051

Additional copies of this manual or other Intel literature may be obtained from:

Literature Department
Intel Corporation
3065 Bowers Avenue
Santa Clara, CA 95051

TABLE OF CONTENTS

INTRODUCTION

Here are the object record formats that define the object language for the 8086 microprocessor. The 8086 object language is the output of all language translators with the 8086 as the target processor. The 8086 object language is input and output for object language processors such as linkers, locaters, librarians, and debuggers.

The 8086 object module formats permit specification of relocatable memory images that may be linked to one another. Capabilities are provided that allow efficient use of the memory mapping facilities of the 8086 microprocessor.

This section defines certain terms fundamental to 8086 R&L. The terms are ordered not alphabetically but so you can read forward without forward references.

DEFINITION of TERMS

- OMF - acronym for Object Module Formats.

- R&L - acronym for Relocation and Linkage.

- MAS - acronym for Memory Address Space. The 8086 MAS is l megabyte (1,048,576). Note that the MAS is distinguished from actual memory, which may occupy only a portion of the MAS.

- MODULE - an “inseparable” collection of object code and other information produced by a translator or by the LINK-86 program. When a distinction must be made,

    - T-MODULE will denote a module created by a translator, such as PLM86 or ASM-86,

    - L-MODULE will denote a module created by (cross) LINK-86 Vl.3 or earlier versions, and

    - R-MODULE will denote a module created by (8086 based) LINK-86 from 1 or more constituent modules. (Note that modules are not “created” in this sense by LOCATE-86; the output module from LOCATE-86 is merely a transformation of the input module.)

  Two observations about modules must be made:

  1) Every module must have a name so that the 8086 Librarian, LIB86, has a handle for the module for display to the user. (If there is no need to provide a handle for LIB86, the name may be

null). Translators will provide names for T-modules, providing a default name (possibly the file name or a null name) if neither source code nor user specifies otherwise.

2) Every T-module in a collection of modules linked together ought to have a different name, so that symbolic debugging systems (such as ICE-86) can distinguish the various line numbers and local symbols. This restriction is not required by R&L, and is not enforced by it.

- LOGICAL SEGMENT - (LSEG) - A contiguous region of memory whose contents are determined at translation-time (except for address-binding). Neither size nor location in MAS are necessarily determined at translation-time: size, although partially fixed, may not be final because the LSEG may be combined at L INK-time to other LSEG's, forming a single LSEG; location in MAS is usually determined at LOCATE-time (although some translators may produce “absolute” object code, whose location is already determined).

- FRAME - A contiguous region of 64K of MAS, beginning on a paragraph boundary (i.e., on a multiple of 16 bytes). This concept is useful because the content of the four 8086 segment resisters defines four (possibly overlapping) FRAME's; no 16-bit address in the 8086 code can access a memory location outside of the current four FRAME's.

  An LSEG is constrained to be no greater than 64K, so that it can fit in a FRAME. This means that any byte in an LSEG may be addressed by a 16-bit offset from the base of a FRAME covering the LSEG.

- PSEG - This term is equivalent to FRAME. Some people prefer “PSEG” to “FRAME” because the terms “PSEG” and “LSEG” reflect the “physical” and “logical” nature of the underlying segments.

- FRAME NUMBER - Every FRAME begins on a paragraph boundary. The “paragraphs” in MAS can be numbered 0,1,2,...,65535. These numbers, each of which defines a FRAME, are called FRAME NUMBERS.

- PARAGRAPH NUMBER - This term is equivalent to “FRAME NUMBER.”

- PSEG NUMBER - This term is equivalent to “FRAME NUMBER.”

- PIC - acronym for Position Independent Code. A PIC module is a module where load addresses and register initialization values are specified relative to segment and group bases. No fixups are allowed.

- LTL - acronym for Load-Time Locatable. An LTL module is like a PIC module except that base fixups are allowed.

- GROUP - a group is a collection of LSEG's defined at translation-time, whose final locations in MAS have been constrained such that there will be at least one FRAME which covers (contains) every LSEG in the collection.

  The notation “Gr A(X,Y,Z)” means that LSEG's X, Y and Z form a group, and that the group's name is A.

  The fact that X, Y and Z are all LSEG's in the same group does not imply any ordering of X, Y and Z in MAS, nor does it imply any contiguity between X, Y and Z.

  In the PIC/LTL case, an LSEG is not allowed to be in more than one group (e.g. defining two groups such as Gr G1(A,C,B) and Gr G2(B,C,D) in the same module is not legal). Otherwise, an LSEG may be in more than one group. The existence of groups such as G1 and G2 is not sufficient to infer that A,B,C,D all lie within some single FRAME, although they might.

- CANONIC - any location in MAS is contained in exactly 4096 distinct FRAME’s; but one of these FRAME’s can be distinguished in that it has a higher FRAME NUMBER than any other FRAME. This distinguished FRAME is called the canonic FRAME of the location.

  Thus, if FOO is a symbol defining a memory location, one may speak of the “canonic FRAME of FOO”, or of “FOO's canonic FRAME”. By extension, if S is any set of memory locations, then there exists a unique FRAME which has the lowest FRAME NUMBER in the set of canonic FRAME's of the locations in S. This unique FRAME is called the canonic FRAME of the set S. Thus, we may speak of the canonic FRAME of an LSEG or of a Group of LSEG's.

- SEGMENT NAME - LSEG's are assigned names at translation time. These names serve only 3 purposes:

  1) They play a role at LINK-time in determining what LSEG's are combined with what other LSEG's.

  2) they may be used at LOCATE-time to designate specific LSEG’s.

  3) they are used in assembly source code to specify groups.

- CLASS NAME - LSEG's may optionally be assigned Class Names at translation-time. Classes define a partition on LSEG’s: two LSEG’s are in the same class if they have the same Class Name.

R&L associates no semantics with specific Class Names; class semantics are completely user-defined. Examples of Class Names might be RED, BLUE, GREEN or ROM, RAM, DISPLAYMEMORY.

The uses of Class Names include the first 2 uses of Segment Names above; additionally, Class Names give the user the power to identify many LSEG's by a single handle at LOCATE-time.

- OVERLAY NAME – LSEG's may optionally be assigned an Overlay Name at translation-time or at LINK-time. This name is specified when the translator or LINK-86 is invoked, and all LSEG's within the same module will be assigned the same Overlay Name.

  An Overlay Name is like a Class Name in that it provides a handle on user-defined equivalence classes of LSEG's. Unlike Class Names, however, Overlay Names have semantics known by the LOCATE-86 program. (In brief, LSEG's in different overlays may be "located" at overlapping MAS locations

- COMPLETE NAME - The "complete name" of an LSEG is defined to be the three-component identification consisting of the Segment Name, Class Name and Overlay Name. LSEG's from different modules will be combined if their Complete Names are identical.

MODULE SEMANTICS

MODULE IDENTIFICATION

To determine that a file contains an object program, a module header record will always be the first record in a module. There are three kinds of header records, and each provides a module name. The additional functions of the header records are explained below.

A module name may be generated during one of two processes: translation or linking. A module that results from translation is called a T-MODULE. A T-MODULE will have a T-MODULE HEADER RECORD (THEADR). A name may be provided in the THEADR record by a translator. This name is then used to identify the source of all symbols and line numbers found in the T-MODULE.

A module that results from linking is called an L-MODULE or an R-MODULE. An L-MODULE will always have an L-MODULE HEADER RECORD (LHEADR). An R-MODULE will always have an R-MODULE HEADER RECORD (RHEADR). In the LHEADR record or the RHEADR record a name may also be provided. This name is available for use as a means of referring to the module without using any of its constituent T-MODULE names. An example would be two T-MODULES, A and B, linked together to form R-MODULE C. R-MODULE C will contain two THEADR records and will begin with an RHEADR record with the name C provided by the linker as a

directive from the user. The R-MODULE C can be referred to by other tools such as the library manager without having to know about the originating module's names, yet the originating module’s names are preserved for debugging purposes.

MODULE ATTRIBUTES

In addition to an optional name, a module may have the attribute of being a main program as well as having a specified starting address. When linking multiple modules together, only one module with the main attribute should be given. The linker EPS specifies the result of finding two or more main modules.

If a module is not a main module yet has a starting address then this value has been provided by a translator, possibly for debugging purposes. A starting address specified for a non-main module could be the entry point of a procedure, which may be loaded and initiated independent of a main program.

In summary, modules may or may not be main as well as may or may not have a starting address.

SEGMENT DEFINITION

A module is defined as a collection of object code defined by a sequence of records produced by a translator. The object code represents contiguous regions of memory whose contents are determined at translation-time. These regions are called LOGICAL SEGMENTS (LSEG's). A module must contain information that defines the attributes of each LSEG. The SEGMENT DEFINITION RECORD (SEGDEF) is the vehicle by which all LSEG information (name, length, memory alignment, etc.) is maintained. The LSEG information is required when multiple LSEG's are combined and when segment addressability (GROUPING, see below) is established. The SEGDEF records are required to follow the first header record (THEADR, or LHEADR, or RHEADR).

SEGMENT ADDRESSING

The 8086 addressing mechanism provides segment base registers from which a 64K byte region of memory, called a FRAME, may be addressed. There is one code segment base register (CS), two data segment base registers (DS, ES), and one stack segment base register (SS).
The possible number of LSEG's that may make up a memory image far exceeds the number of available base registers. Thus, base registers may require frequent loading. This would be the case in a modular program with many small data and/or code LSEG's.

Hence the motivation to collect LSEG's together to form one addressable unit that can be contained within a memory frame. The name for this

addressable unit is a GROUP and has been defined earlier in the DEFINITION OF TERMS.

To allow addressability of objects within a GROUP to be established, each GROUP must be explicitly defined in the module. The GROUP DEFINITION RECORD (GRPDEF) provides a list of constituent segments either by segment name or by segment attribute such as "the segment defining symbol Foo" or "the segments with class name ROM".

The GRPDEF records within a module must follow all SEGDEF records as GRPDEF records may reference SEGDEF records in defining a GROUP. The GRPDEF records must also precede all other records but header records as some R&L products must process them first. The explicit ordering of records is given later.

SYMBOL DEFINITION

Within a module there may be six different types of symbol definition records. The necessity for these records is based on two requirements: 1) references to externally defined symbols should be resolved by equivalently defined symbols in another module (linking) and 2) attributes of locally defined symbols and line numbers should be made available for debugging purposes.

The requirements for symbol definition records for module linking is satisfied by the PUBLIC NAMES DEFINITION RECORD (PUBDEF), the EXTERNAL NAMES DEFINITION RECORD (EXTDEF), and the TYPE DEFINITION RECORD (TYPDEF). Their semantics will be explained later.

The requirements for debugging information are satisfied by the LOCAL SYMBOLS RECORD (LOCSYM), the LINE NUMBERS RECORD (LINNUM), the DEBUG SYMBOLS RECORD (DEBSYM), the BLOCK DEFINITION RECORD (BLKDEF), the BLOCK END RECORD (BLKEND), and the TYPE DEFINITION RECORD (TYPDEF). The association of the line numbers and local symbols to their original defining modules is essential and maintained by the THEADR record as explained earlier.

DATA

The data that defines the memory image represented by a module is maintained in six varieties of DATA records. The DATA records are of three classes: relocatable, physical and logical.

There are two Relocatable DATA records: RELOCATABLE ENUMERATED DATA RECORD (REDATA) and RELOCATABLE ITERATED DATA RECORD (RIDATA).
Each relocatable DATA record is associated with a SEGDEF record or a FRAME number, and perhaps a GRPDEF Record. The SEGDEF record or the FRAME number, and the GRPDEF record provide information to determine the absolute address at which the data bytes are to be loaded. The RIDATA record differs in that the data bytes are represented within a

structure that must be expanded by the loader. The purpose of the RIDATA record is to reduce module size by encoding repeated data rather than explicitly enumerating each byte, as the REDATA record does.

There are two Physical DATA records: PHYSICAL ENUMERATED DATA RECORD (PEDATA) and PHYSICAL ITERATED DATA RECORD (PIDATA). The PEDATA and PIDATA records provide an absolute address at which the data bytes it contains are to be loaded.

There are also two logical DATA record LOGICAL ENUMERATED DATA RECORD (LEDATA) and LOGICAL ITERATED DATA RECORD (LIDATA). Each logical DATA record is associated with a SEGDEF record. The SEGDEF record provides information that allows the logical DATA record is to be converted to either Relocatable DATA records or Physical DATA records.

Data bytes for all LSEG's are maintained in logical DATA records, as an LSEG is either relocatable or it has been assigned an address (absolute) but has not been divorced from GROUP information.

In summary, there are three classes of DATA records, RELOCATABLE, PHYSICAL, and LOGICAL. The data bytes of the "unnamed absolute segment", divorced from all LSEG and GROUP information, are found in PHYSICAL DATA RECORDS. Data bytes from all LSEG's, absolute or relocatable, are found in LOGICAL DATA RECORDS. The ENUMERATED and ITERATED attributes within the classes are two ways of representing the actual data bytes.

An 8086 loader can load RDATA or PDATA records but will probably not be able to maintain the LSEG table information required for loading LDATA Records. Thus, Relocatable and Physical DATA records are sometimes called "Loadable" DATA records, and Logical DATA records are called "Non-Loadable" DATA records.

INDICES

Throughout the 8086-OMF specification, "index" fields occur. An index is an integer that selects some particular item from a collection of such items. (Exhaustive list of examples: NAME INDEX, SEGMENT INDEX, GROUP INDEX, EXTERNAL INDEX, TYPE INDEX, BLOCK INDEX.)

> **(Note)** An index is normally a positive number. The index value zero is reserved and may carry a special meaning dependent upon the type of index (e.g., a Segment Index of zero specifies the "Unnamed, absolute pseudo-segment; a Type Index of zero specifies the "Untyped type" (which is different from "Decline to state")). **(End of Note)**

In general, indices must assume values quite large (i.e., much larger than 255). Nevertheless, a great number of object files will contain

no indices with values greater than 50 or 100. Therefore, indices will be encoded in 1 or 2 bytes, as required:

The high-order (left-most) bit of the first (and possibly the only) byte determines whether the index occupies one byte or two. If the bit is 0, then the index is a number between 0 and 127, occupying one byte. If the bit is 1, then the index is a number between 0 and 32K-1, occupying two bytes, and is determined as follows: the low-order 8 bits are in the second byte, and the high-order 7 bits are in the first byte.

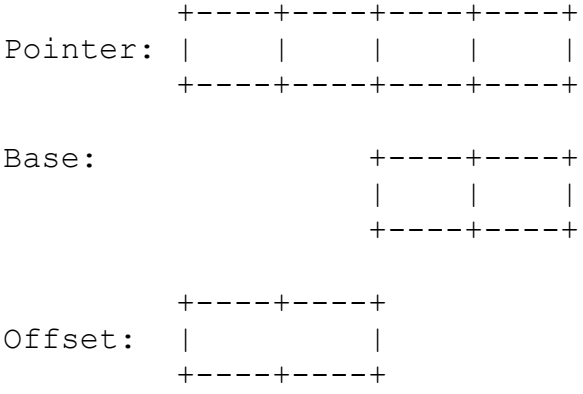
CONCEPTUAL FRAMEWORK for FIXUP's

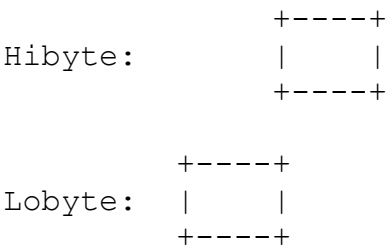
A “Fixup” is some modification to object code, requested by a translator, performed by the R&L system, achieving address binding. (see Appendix 4 for Examples)

> **(Note)** This definition of “fixup” accurately represents the viewpoint maintained by the R&L system. Nevertheless, the R&L system can be used to achieve modifications of object code (i.e., “fixups”) that do not conform to this definition. For example, the binding of code to either of hardware floating point or software floating point subroutines, is a modification to an operation code, where the operation code is treated as if it were an address. The above definition of “fixup” is not intended to disallow or disparage object code modifications in the wider sense. **(End of Note)**

8086 and/or 8089 translators specify fixup by giving four data: (1) the plate and type of a LOCATION to be fixed up, (2) one of the two possible fixup MODE’s, (3) a TARGET, which is a memory address to which LOCATION must be made to refer, and (4) a FRAME defining a context within which the reference takes place.

- LOCATION – There are 5 types of LOCATION: a POINTER, a BASE, an OFFSET, a HIBYTE, and a LOBYTE:





The vertical alignment of this diagram illustrates 4 points (remember that the high order byte of a word in 8086 memory is the byte with the higher address): (1) a BASE is merely the high order word of a pointer (and R&L doesn't care if the low order word of the pointer is present or not); (2) an OFFSET is merely the low order word of a pointer (and R&L doesn't care if the high order word fo1 1ows or not); (3) a HIBYTE is merely the hiqh order half of an OFFSET (and R&L doesn't care if the low order half precedes or not); (4) a LOBYTE is merely the low order half of an OFFSET (and R&L doesn't care if the high order half follows or not).

A LOCATION is specified by 2 data: (1) which of the above 5 types the LOCATION is, and (2) where the LOCATION is. (1) is specified by the LOC subfield of the LOCAT field of the FIXUPP Record; (2) is specified by the DATA RECORD OFFSET subfield of the LOCAT field of the FIXUPP Record.

- MODE - R&L supports 2 kinds of fixups: “self-relative” and “segment-relative”.

Self-relative fixups support the 8- and 16-bit offsets that are used in the CALL, JUMP and SHORT-JUMP instructions. Segment-relative fixups support all other addressing modes of the 8086.

- TARGET - The TARGET is the location in MAS being referenced. (More explicitly, the TARGET may be considered to be the lowest byte in the object being referenced.) A TARGET is specified in one of 8 ways. There are 4 “primary” ways, and 4 “secondary” ways. Each primary way of specifying a TARGET uses 2 data: an INDEX-or-FRAME-NUMBER ‘X’, and a displacement ‘D’:

  (T0) X is a SEGMENT INDEX. The TARGET is the D’th byte in the LSEG identified by the INDEX.

  (Tl) X is a GROUP INDEX. The TARGET is the D’th byte following the first byte in the LSEG in the group that is eventually LOCATE’d lowest in MAS.

  (T2) X is an EXTERNAL INDEX. The TARGET is the D’th byte in the FRAME identified by the FRAME NUMBER (i.e., the address of TARGET is (X*16)+D).

Each secondary way of specifying a TARGET uses only 1 datum: the INDEX-or-FRAME-NUMBER X. An implicit displacement equal to zero is assumed:

(T4) X is a SEGMENT INDEX. The TARGET is the 0'th (first) byte in the LSEG identified by the INDEX.

(T5) X is a GROUP INDEX. The TARGET is the 0'th (first) byte in the LSEG in the specified group that is eventually LOCATE'd lowest in MAS.

(T6) X is an EXTERNAL INDEX. The TARGET is the byte whose address is (eventually given by) the External Name identified by the INDEX.

(T7) X is a FRAME NUMBER. The TARGET is the byte whose 20-bit address is (X*16).

The following nomenclature is used to describe a TARGET:

```
TARGET: SI (<segment name>),<displacement>   [T0]
TARGET: GI (<group name>),<displacement>     [Tl]
TARGET: EI (<symbol name>),<displacement>    [T2]
TARGET: <FRAME NUMBER>,<displacement>        [T3]
TARGET: SI (<segment name>)                  [T4]
TARGET: GI (<group name>)                    [T5]
TARGET: EI (<symbol name>)                   [T6]
TARGET: <FRAME NUMBER>                       [T7]
```

Here are some examples of how this notation can be used:

TARGET: SI(CODE),1024	The 1025th byte in the segment "CODE".
TARGET: GI(DATAAREA)	The location in MAS of a group called "DATAAREA".
TARGET: EI(SIN)	The address of the external subroutine "SIN".
TARGET: 8000H,24H	MAS location 80024H.
TARGET: EI(PAYSCHEDULE),24	the 24th byte following the location of an EXTERNAL data structure called "PAYSCHEDULE".

Although TARGET: SI(A)" and "TARGET: SI(A),0" both specify the same TARGET, their use can have different effects, as is discussed below in the section on intermediate values in fixup arithmetic.

- FRAME - Every 8086 memory reference is to a location contained within some FRAME; where the FRAME is designated by the content of some segment register. In order for R&L to form a correct, usable memory reference, it must know not only what the TARGET is, but also with respect to which FRAME the reference is being made. Thus every fixup specifies such a FRAME, in one of 6 ways (F0,...,F5) described below. Some ways use a datum, X, which is an INDEX-or-FRAME-NUMBER, as above. Other ways require no datum.

  This not the case of an 8089 self-relative reference. The reference may be to any location within an 8089 program, of FRAME. The only restriction is that the displacement between the LOCATION and the TARGET must be within 32K. To indicate this type of fixup, a 7th way (F6) of specifying a frame is introduced.

  Below is the description of the seven ways of specifying frames:

  (F0) X is a SEGMENT INDEX. The FRAME is the canonic FRAME of the LSEG defined by the INDEX.

  (F1) X is a GROUP INDEX. The FRAME is the canonic FRAME defined by the group (i.e., the canonic FRAME defined by the LSEG in the group that is eventually LOCATE'd lowest in MAS).

  (F2) X is an EXTERNAL INDEX. The FRAME is determined when the External Name's public definition is found. There are 3 cases:

    (F2a) The symbol is defined relative to some LSEG, and there is no associated Group. The LSEG's canonic FRAME is specified.

    (F2b) The symbol is defined absolutely, without reference to an LSEG, and there is no associated Group. The FRAME is specified by the FRAME NUMBER subfield of the PUBDEF Record (q.v.) that gives the symbol's definition.

    (F2c) Regardless of how the symbol is defined, there is an associated Group. The canonic FRAME of the Group is specified. (The group is specified by the GROUP INDEX subfield of the PUBDEF Record (q.v.).)

  (F3) X is a FRAME NUMBER (specifying the obvious FRAME).

  (F4) No X. The FRAME is the canonic FRAME of the LSEG containing LOCATION. (If LOCATION is specified absolutely (i.e., in a PEDATA Record or a PIDATA Record (q.v.)), then it is not "contained" in

an LSEG; in this case the FRAME is determined as in (F2) above, taking the FRAME NUMBER from the FRAME NUMBER field of the DATA Record.

(F5) No X. The FRAME is determined by the TARGET. There are 4 cases:

(F5a) The TARGET specified a SEGMENT INDEX: in this case, the FRAME is determined as in (F0) above.

(F5b) The TARGET specified a GROUP INDEX: in this case, the FRAME is determined as in (Fl) above.

(F5c) The TARGET specified an EXTERNAL INDEX: in this case, the FRAME is determined as in (F2) above.

(F5d) The TARGET is specified with an explicit FRAME NUMBER: in this case, the FRAME is determined as in (F3) above.

(F6) No X. There is no FRAME. This is a way to indicate to R&L that an 8089 self-relative reference is to be processed. A signed displacement between the LOCATION 20-bit address and the TARGET 20-bit address must be computed.

Nomenclature describing FRAME's is similar to the above nomenclature for TARGET's, viz:

```
FRAME: SI(<segment name>)     [F0]
FRAME: GI(<group name>)       [F1]
FRAME: EI(<symbol name>)      [F2]
FRAME: <FRAME NUMBER>         [F3]
FRAME: LOCATION               [F4]
FRAME: TARGET                 [F5]
FRAME: NONE                   [F6]
```

In practice, for an 8086 memory reference, it is likely that the FRAME specified by a self-relative reference will be the canonic FRAME of the LSEG containing the LOCATION, and the FRAME specified by a segment relative reference will be the canonic FRAME of the LSEG containing the TARGET. This will be further explained below.

SELF-RELATIVE FIXUPS

A self-relative fixup operates as follows: A memory address is implicitly defined by LOCATION: namely the address of the byte following LOCATION (because at the time of a self-relative reference, the 8086 IP (Instruction Pointer) or the 8089 TP (Task block Program pointer) is pointing to the byte following the reference).

For 8086 self-relative references, if either LOCATION or TARGET are outside the specified FRAME, R&L gives a warning. Otherwise, there is a unique 16-bit displacement which, when added to the address implicitly defined by LOCATION, will yield the relative position of TARGET in the FRAME.

For 8089 self-relative references (F6), if TARGET is not within 32K from LOCATION, R&L gives a warning. Otherwise, there is a unique 16-bit signed displacement between the LOCATION and the TARGET.

If the LOCATION is an OFFSET, the displacement is added to LOCATION modulo 65536: no errors are reported.

If the LOCATION is a LOBYTE, the displacement must be within the range {-128:127}, otherwise R&L will give a warning. The displacement is added to LOCATION modulo 256.

If the LOCATION is a BASE, POINTER, or HIBYTE, it is unclear what the translator had in mind, and the action taken by R&L is defined by LINK-86 and/or LOCATE-86 EPS's.

SEGMENT-RELATIVE FIXUPS

A segment-relative fixup operates in the following way: a non-negative 16-bit number, FBVAL, is defined as the FRAME NUMBER of the FRAME specified by the fixup, and a signed 20-bit number, FOVAL, is defined as the distance from the base of the FRAME to the TARGET. If this signed 20-bit number is less than 0 or greater than 65535, then R&L will report an error. Otherwise FBVAL and FOVAL are used to fixup LOCATION in the following fashion:

(1) if LOCATION is a POINTER, then FBVAL is added (modulo 65536) to the high order word of POINTER, and FOVAL is added (modulo 65536) to the low order word of POINTER.

(2) if LOCATION is a BASE, then FBVAL is added (modulo 65536) to the BASE; FOVAL is ignored.

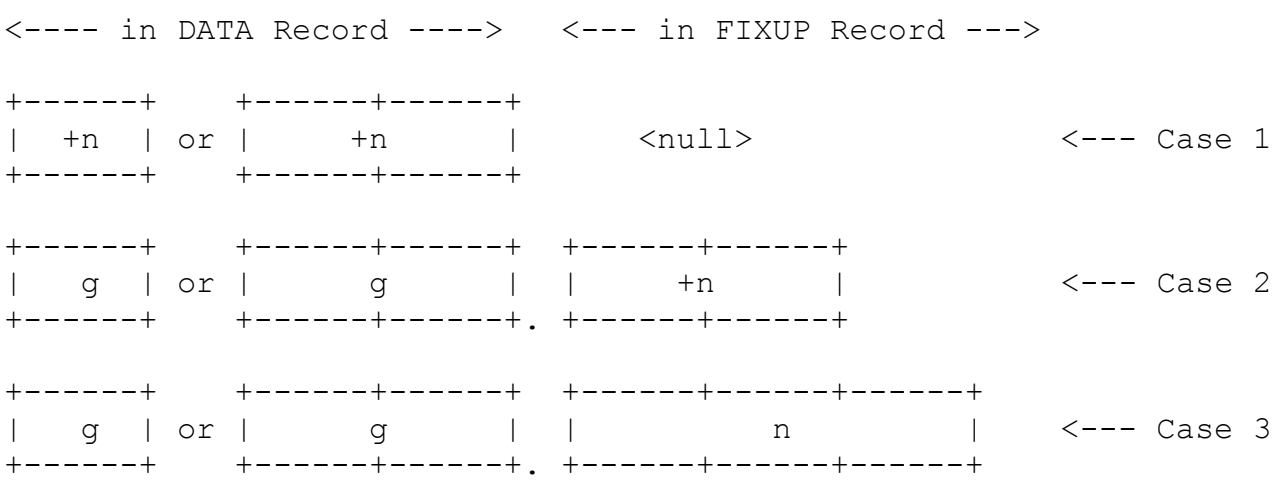
(3) if LOCATION is an OFFSET. then FOVAL is added (modulo 65536) to the OFFSET; FBVAL is ignored.

(4) if LOCATION is a HIBYTE, then (FOVAL / 256) is added (modulo 256) to the HIBYTE; FBVAL is ignored. (The indicated division is "integer division", i.e., the remainder is discarded.)

(5) if LOCATION is a LOBYTE, then (FOVAL modulo 255) is added (modulo 256) to the LOBYTE; FBVAL is ignored.

INTERMEDIATE VALUES in FIXUP ARITHIMETIC

The 8086 Object Module Formats guarantee fixups in the sense that, if a TARGET cannot be accessed from a LOCATION with the assumed FRAME, then that failure can be detected and R&L can issue a warning message. This checking is called “access verification”. In order to perform this checking, LINK-86 and LOCATE-86 need to retain intermediate values of its address arithmetic. These intermediate values are retained either in the DATA Record, or in the FIXUP Record. The following diagram illustrates three cases:



Case 1 illustrates the situation where a fixup is specified in a “secondary” way. No explicit displacement ‘D’ is provided in the FIXUP Record, so arithmetic must be done in the LOCATION itself, in the DATA Record. As the diagram shows, the LOCATION may be a byte or a word. (If LOCATION is a POINTER, arithmetic is on each half separately, so the above diagram applies separately to each half of a POINTER.) In Case 1, the value(s) in LOCATION are considered to be non-negative numbers (“+n”), and are considered to be equivalent to a specification of a displacement ‘D’; thus the R&L access verification incorporates the value “+n”.

Case 2 illustrates the situation where a fixup is specified in a “primary” way. An explicit displacement ‘D’ is provided in the FIXUP Record. This displacement is considered to be a non-negative number (“+n”). When all arithmetic required by the fixup is complete, the resultant value (in the FIXUP Record) is checked for validity by R&L, and then, finally, that result is added (modulo 256 or modulo 65536) to the original content of LOCATION (“g”). The value “g” may be considered as non-negative, or as signed 2's complement; R&L doesn't care because there is no checking in this final stage of the fixup.

Case 3 is the same as Case 2, except that the displacement 'D', instead of being restricted to non-negative numbers in the range {0:65535}, may represent signed (2's complement) numbers in the range {-

1,048,576:1,048,575}. (Note: initially, this case will not be supported. It is designed into the formats for completeness: it allows support, with R&L access verification, of TARGET'S specified in a "primary" way, with negative displacements 'D'.)

Here are some cases where a "primary" specification of a TARGET is necessary or desirable:

First, yet another definition: a "REFERENT" is a memory location, with respect to which a TARGET is positioned. This is best made clear by an example: in the specification:

```
TARGET: EI(STRUCT),24
```

the TARGET is the 24'th byte after the location named "STRUCT"; the REFERENT is the location named "STRUCT" itself.

(1) A SHORT-JMP is being made to an external subroutine. In this case, the TARGET should be specified as:

```
TARGET: EI(subroutine),0000H
```

The reason is that when LINK-86 learns where the subroutine is located, it will probably be a known offset (dl) within some LSEG A. Thus, LINK-86 will convert the above TARGET to the form:

```
TARGET: SI(A),dl
```

Now the programmer may be correct in "knowing" that when the program is eventually LOCATE'd, the TARGET will be within 128 bytes of LOCATION; however, this does not mean that dl is less than l28! Thus, as LINK-86 maintains the (possibly changing) value of dl as various pieces of LSEG A are combined, it needs a full word to maintain the offset. Since the LOCATION is a single byte, the translator must provide an offset field in the fixup record itself for LINK-86 to maintain intermediate fixup values.

(2) The translator wishes to reference "backwards" from the REFERENT. For example, if the TARGET is the word in front of the external array ARY, and the reference is with respect to a base register that will contain the address of LSEG named FOO, the translator would use:

```
FRAME:  SI(FOO)
TARGET: EI(ARY),0000H
```

and place the "negative offset" FFFEH in LOCATION. R&L will perform access verification to the REFERENT ARY: however, access to the TARGET is not guaranteed, and is the programmer's responsibility.

Note: if Case 3 in the above diagram were available, the translator could use:

```
FRAME: SI(FOO)
TARGET: EI(ARY),-2
```

and R&L would perform access verification, not to the REFERENT ARY (as above), but to the actual TARGET (in front of ARY)!

(2) (continued) The calculation by LOCATE-86 involves 3 quantities: the MAS-location of FOO, the MAS-location of the LSEG (say, BAZ) containing ARY, and the relative offset of ARY within BAZ. LOCATE-86 can enforce that the final offset, which is the difference (location of SAZ plus relative offset) - (location of FOO) is not greater than 65535 provided that all quantities entering into this difference are known. If the translator had specified the fixup as:

```
FRAME: SI(FOO)
TARGET: EI(ARY)
```

then LINK-86 would have had to maintain the (possibly changing from linkage to linkage) relative offset of ARY within BAZ. in the LOCATION itself, where it gets “added” to the content FFFEH. And because the R&L system cannot know if the FFFEH was a negative 2 or a positive 65534, the access verification of R&L may thwart the translator's intentions.

The following example (3) is a case where access verification works whether the TARGET specification is “primary” or “secondary”:

(3) The translator wishes to reference “forwards” from a REFERENT, and to ensure that the TARGET lies within the specified FRAME. For example, we wish to reference the l00'th byte in an external structure STRCT. The translator may specify the fixup as:

```
FRAME: SI(FOO)
TARGET: EI(STRCT),99
```

R&L will ensure that the distance from the canonic FRAME of FOO to the 100'th byte of STRCT is less than 65536. (Note that this constraint might be achieved even if STRCT lies outside the canonic FRAME of FOO.)

(4) Hibyte fixups specified in a primary way will be correct in that a full word is used to accumulate the value of an offset. Only after LOCATE'inq will the value of the hibyte of an offset be used as a fixup value. This prevents the loss of accuracy due to truncation of low byte before adding the address at which an object is LOCATE'd.

MODULE SYNTAX

RECORD ORDER

A object code file must contain a sequence of (one or more) modules, or a library containing zero or more modules. A module is defined as a collection of object code defined by a sequence of object records. The following syntax shows the valid orderings of records to form a module. In addition, the given semantic rules provide information about how to interpret the record sequence. The syntactic description language used herein is defined in WIRTH: CACM, November 1977, v 20, n 11, p 822 - 823.

```
object_file    = sequence | library.
sequence       = module {module}.
library        = LIBHED {module} libtail.
module         = tmod | lmod | rmod | omod.
tmod           = THEADR sgr_table {component} modtail.
lmod           = LHEADR sgr_table {data} {t_component} modtail.
rmod           = RHEADR sgr_table {data} {t_component} modtail.
omod           = RHEADR sgor_table {o_component} o_modtail.
sqr_table      = seg_grp [REGINT].
sqor.table     = seg_grp {OVLDEF} [REGINT].
seg.qrp        = {LNAMES} {SEGDEF} { TYPDEF | EXTDEF | GRPDEF }.
o_component    = {data} {t_component} ENDREC.
t_component    = THEADR {component}.
component      = data | debuq_record.
data           = content_def | thread_def | TYPDEF | PUBDEF | EXTDEF.
debuq_record   = LOCSYM | LINNUM | DEBSYM | BLKDEF | BLKEND | ENDREC.
content_def    = data_record {FIXUPP}.
thread_def     = FIXUPP. (containing only thread fields)
data_record    = LIDATA | LEDATA | PIDATA | PEDATA | REDATA | RIDATA.
o_modtail      = {OVLDEF} modtail.
mod_tail       = [REGINT] MODEND.
libtail        = LIBNAM LIBLOC LIBDIC.
```

See also Syntax Diagrams.

NOTE: The character strings represented by capital letters above are not literals but are identifiers that are further defined in the section defining the Record Formats.

The following rules apply:

1. A FIXUPP record always refers to the previous DATA record.

2. The debug records have as their originating module the module named by the nearest preceding THEADR record.

3. All LNAMES, SEGDEF, GRPDEF, TYPDEF, and EXTDEF records must precede all records that refer to them.

4. COMENT records may appear anywhere within a file, except as the first or last record in a file or module, within a content_def, or within a libtail.

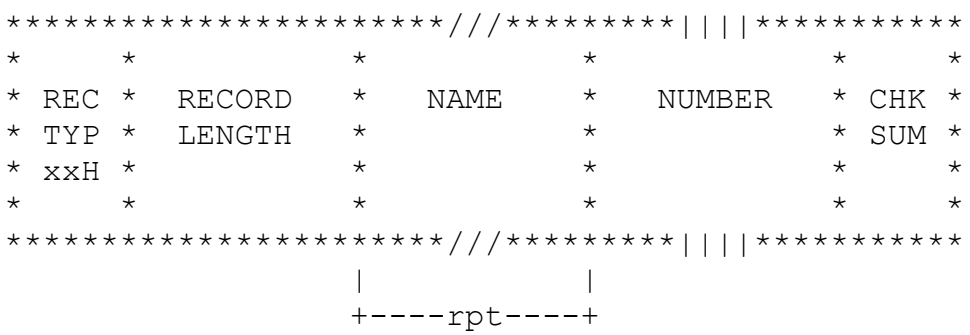
5. OVLDEF records may appear either immediately after the segment and group definitions or at the end (before the REGINT and MODEND records), but not at both places. The number of OVLDEF records must be equal to the number of o_components, and the order of these records must be same as the o_component order, the first OVLDEF record pointing to the 'root' part.

6. As with the OVLDEF records, the REGINT record may appear either at the beginning of a module (after SEGDEF's, GRPDEF's, and OVLDEF's if any) or at the end (before the MODEND record), but there cannot be two REGINT records in the same module.

INTRODUCTION to the RECORD FORMATS

The following pages present diagrams of Record Formats in schematic form. Here is a sample, to illustrate the various conventions:

SAMPLE RECORD FORMAT
(SAMREC)



TITLE and OFFICIAL ABBREVIATION

At the top is the name of the Record Format Described, together with an official abbreviation. To promote uniformity among various programs, including translators, debuggers, the various R&L products, and various tools such as EDOJ86 and OJED86, the abbreviation should be used in both code and documentation. The abbreviation is always 6 letters.

The BOXES

Each format is drawn with boxes of two sizes. The narrow boxes, outlined entirely with asterisks, represent single bytes. The wide boxes, outlined entirely with asterisks, represent two bytes each. The wide boxes, outlined with asterisks, but with three slashes in the top and bottom, represent a variable number of bytes, one or more, depending upon content. The wide boxes, outlined with asterisks, but with four vertical bars in the top and bottom, represent 4-byte fields.

REC TYP

The first byte in each record contains a value between 0 and 255, indicating which record type the record is.

RECORD LENGTH

The second field in each record contains the number of bytes in the record, exclusive of the first 2 fields.

NAME

Any field that indicates a “NAME” has the following internal structure: the 1st byte contains a number between 0 and 40, inclusive, that indicates the number of remaining bytes in the field. The remaining bytes are interpreted as a byte string; each byte must represent the Ascii code of a character drawn from this set:
{ ?@:._0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ }. Most translators will choose to constrain the character set more strictly; the above set has been chosen to “cover” that required by all current processors.

NUMBER

A 4-byte NUMBER field represents a 32-bit unsigned integer, where the first 8 bits (least-significant) are stored in the first byte (lowest address), the next 8 bits are stored in the second byte, etc.

REPEATED OR CONDITIONAL FIELDS

Some portions of a Record Format contain a field or series of fields that may be repeated 0 or more times. Such portions are indicated by the “repeated” or “rpt” brackets below the boxes.

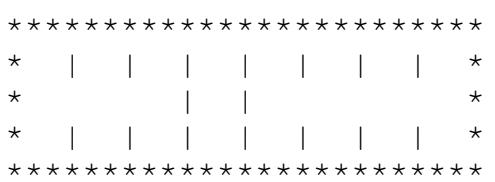
Similarly, some portions of a Record Format are present only if some given condition is true; these fields are indicated by similar “conditional” or “cond” brackets below the boxes.

CHK SUM

The last field in each record is a check sum, which contains the 2's complement of the sum (modulo 256) of all other bytes in the record. Therefore, the sum (modulo 256) of all bytes in the record equals 0.

BIT FIELDS

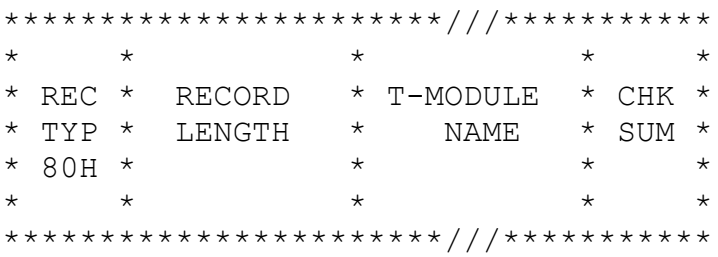
Descriptions of contents of fields will sometimes get down to the bit level. Boxes outlined in asterisks, but with vertical lines drawn through them, represent bytes or words; the vertical lines indicate bit boundaries, thus the byte, represented below, has 3 bit-fields of 3-, 1-, and 4-bits:



RECORD FORMATS

T-MODULE HEADER RECORD

(THEADR)



Every module output from a translator must have a T-MODULE HEADER RECORD. Its purpose is to provide the identity of the original defining module for all line numbers and local symbols encountered in the module up to the following T-MODULE HEADER RECORD or MODULE END RECORD.

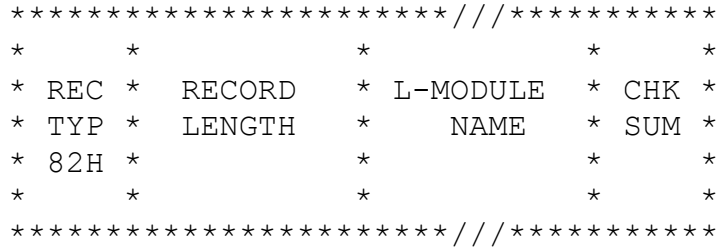
This record can also serve as the header for a module i.e., it can be the first record, and will be for modules output from translators.

T-MODULE NAME

The T-MODULE NAME provides a name for the T-Module.

L-MODULE HEADER RECORD

(LHEADR)



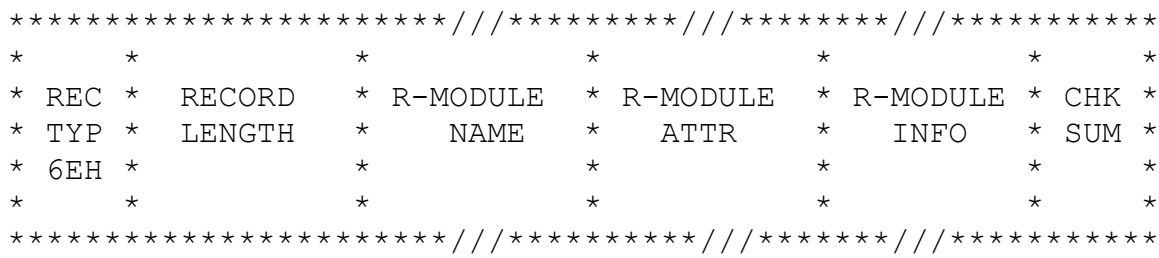
Every module previously created by (cross) LINK-86 (Vl.3 or earlier) or by LOCATE-86 may have an L-MODULE HEADER RECORD. This record serves only to identify a module that has been processed (output) by LINK-86/LOCATE-86. When several modules are linked to form another module, the new module requires a name, perhaps unique from those of the linked modules, by which it can be referred to (by the LIB86 program, for example).

L-MODULE NAME

The L-MODULE NAME provides a name for the L-Module.

R-MODULE HEADER RECORD

(RHEADR)



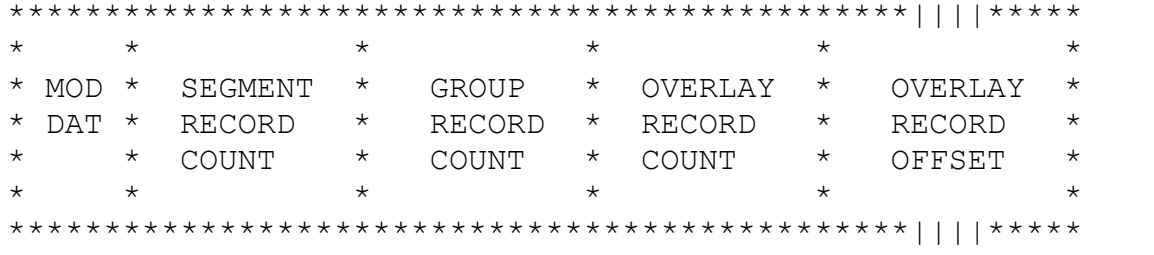
Every module created by LINK-86/LOCATE-86 may have an R-MODULE HEADER RECORD. This record serves to identify a module that has been processed (output) by LINK-86/LOCATE-86. It also specifies the module attributes and gives information on memory usage and need. When several modules are linked to form another module, the new module requires a name, perhaps unique from those of the linked modules, by which it can be referred to (by the LIB86 program, for example).

R-MODULE NAME

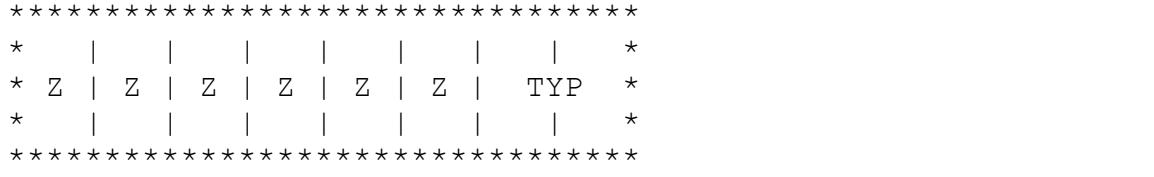
The R-MODULE NAME provides a name for the R-Module.

R-MODULE ATTR

The R-MODULE ATTR field provides information on various module attributes, and has the following format:



The MOD DAT subfield has the following format:



Z’s indicates that these 1-bit fields have not currently been assigned a function. These bits are required to be zero.

TYP is a 2-bit subfield that specifies the module type. The semantics are defined as follows:

TYP=0 The module is an absolute module.
TYP=1 The module is a relocatable module. Fixups other than base fixups may still be present.
TYP=2 The module is a Position Independent Code module. It can be loaded anywhere. No fixups are needed.
TYP=3 The module is a Load-Time Locatable Module. It can be loaded anywhere with perhaps some base fixups to be performed.

The SEGMENT RECORD COUNT subfield indicates the number of Segment Definition Records in the module.

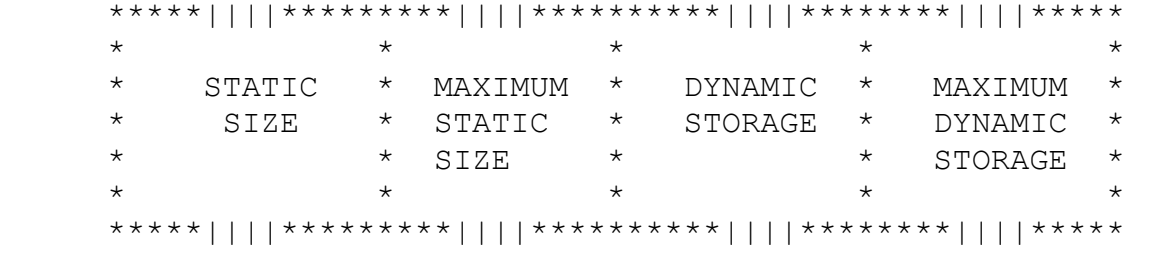
The GROUP RECORD COUNT subfield indicates the number of Group Definition Records in the module.

The OVERLAY RECORD COUNT subfield indicates the number of Overlay Definition Records in the module (including Overlay Definition Record for the 'Root').

The OVERLAY RECORD OFFSET subfield is a 4-byte field. It contains a 32-bit unsigned number indicating the location in bytes, relative to the start of the object file, of the first Overlay Definition Record in the module. This field must be zero when OVERLAY RECORD COUNT is zero.

R-MODULE INFO

The R-MODULE INFO field contains a sequence of four 32-bit unsigned numbers specifying the different types and sizes (in bytes) of memory space that the module will need. It has the following format:



STATIC SIZE is the total size of the LTL segments in the module. This is the minimum static memory space that must be allocated to the module so that the module can be loaded.

MAXIMUM STATIC SIZE is the maximum total size of the LTL segments in the module. This value must be greater than or equal to STATIC SIZE. (By default MAXIMUM STATIC SIZE is set equal to STATIC SIZE) This value only gives the maximum space needed. Depending on available memory, the loader may

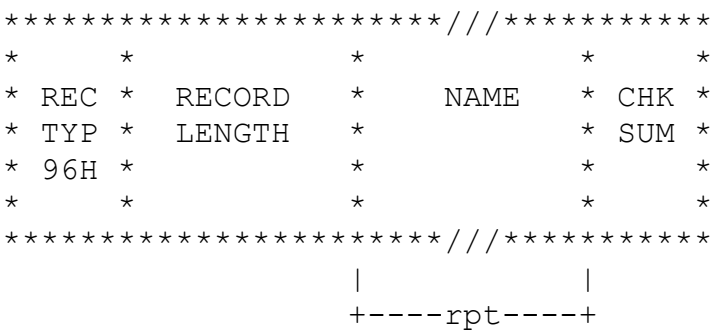
allocate any value between the STATIC SIZE and the MAXIMUM STATIC SIZE.

DYNAMIC STORAGE is the memory space that must be allocated (for buffer, for dynamic expansion, etc...) at load-time. The default value is zero.

MAXIMUM DYNAMIC STORAGE is the maximum dynamic memory that might be needed by the module. This value must be greater than or equal to DYNAMIC STORAGE (By default MAXIMUM DYNAMIC STORAGE value is set equal to DYNAMIC STORAGE value).

LIST OF NAMES RECORD

(LNAMES)



This Record provides a list of Names that may be used in following SEGDEF and GRPDEF Records as the names of Segments, Classes, Overlays and/or Groups.

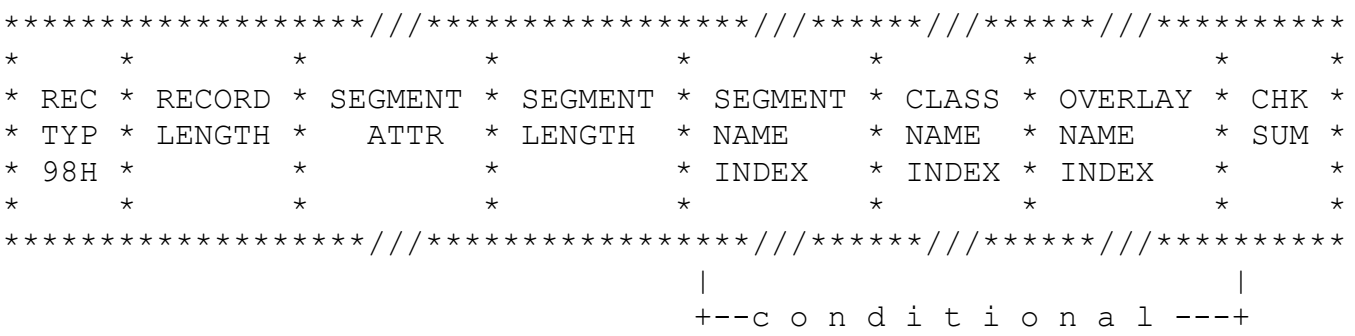
The ordering of LNAMES Records within a module, together with the ordering of Names within each LNAMES Record, induces an ordering on the Names. Thus, these names are considered to be numbered: 1, 2, 3, 4, ... These numbers are used as “Name Indices” in the Segment Name Index, Class Name Index, Overlay Name Index and Group Name Index fields of the SEGDEF and GRPDEF Records.

NAME

This repeatable field provides a name, which may have zero length.

SEGMENT DEFINITION RECORD

(SEGDEF)



SEGMENT INDEX values 1 through 32767, which are used in other record types to refer to specific LSEG's, are defined implicitly by the sequence in which SEGDEF Records appear in the object file. (SEGMENT INDEX 0 is reserved to indicate the “unnamed absolute segment”, which is not really a segment: it is a possibly empty set of possibly disjoint regions of memory; it is normally created by LOCATE-86, although translators may create portions of it as well, if they wish.)

SEG ATTR

The SEG ATTR field provides information on various attributes of the segment, and has the following format:

The ACBP byte contains 4 numbers, and the A, C, B and P attribute specifications. This byte has the following format:

A (Alignment) is a 3-bit subfield that specifies the alignment attribute of the LSEG. The semantics are defined as follows:

A=0 SEGDEF describes an absolute LSEG.
A=1 SEGDEF describes a relocatable, byte aligned LSEG.

A=2 SEGDEF describes a relocatable, word aligned LSEG.
A=3 SEGDEF describes a relocatable, paragraph aligned LSEG.
A=4 SEGDEF describes a relocatable, page aligned LSEG.
A=5 SEGDEF describes an unnamed absolute portion of MAS.
A=6 SEGDEF describes a load-time locatable (LTL), paragraph aligned LSEG if not member of any group.

In addition the value of A determines if one or several "conditional" fields will be present. If A=0 or A=5 then the FRAME NUMBER and OFFSET fields will be present. If A=6 then the LTL DAT, MAXIMUM SEGMENT LENGTH, and GROUP OFFSET fields will be present. If A<>5 then the three NAME INDEX fields will be present.

C (Combination) is a 3-bit subfield that specifies the combination attribute of the LSEG. Absolute segments (A=0 or A=5) must have combination zero (C=0). In this case the segments will be combined like C=6 below if and only if their FRAME NUMBER's and OFFSET's match (For A=0 their complete names must match as well). For relocatable segments, the C field encodes a number 0,1,2,4,5,6 or 7 indicating how the segment may be combined. The interpretation of this attribute is best given by considering how two LSEG's are combined: Let X,Y be LSEG's, and let Z be the LSEG resulting from the combination of X,Y. Let LX and LY be the lengths of X and Y, and let MXY denote the maximum of LX,LY. Let G be the length of any gap required between the X- and Y-components of Z to accommodate the alignment attribute of Y. Let LZ denote the length of the (combined) LSEG Z; let dx (0<=dx<LX) be the offset in X of a byte, and let dy similarly be the offset in Y of a byte. Then the following table gives the length LZ of the combined LSEG Z, and the offsets dx' and dy' in Z for the bytes corresponding to dx in X and dy in Y:

C	LZ	dx'	dy'
2	LX+LY+G	dx	dy+LX+G
4	LX+LY	dx	dy
5	LX+LY	dx+LY	dy+LX
6	MXY	dx	dy
7	MXY	dx+MXY-LX	dy+MXY-LY

The above table has no lines for C=0, C=1 or C=3. C=0 indicates that the relocatable LSEG may not be combined; C=1 has the same combination semantics as C=6, but additionally "distinguishes" the LSEG so that LOCATE-86 will (in the default case) place the LSEG above all other LSEG's in MAS (this corresponds to the MEMORY segment semantics of 8080 R&L); C=3 is undefined.

B (Big) is a 1-bit subfield which, if 1, indicates that the Segment Length is exactly 64K (65536). In this case the SEGMENT LENGTH field must contain zero.

P (Page-Resident) is a l-bit subfield which, if 1, demands that the segment be located in MAS without crossing a page boundary. (This corresponds to the “in-page” relocation type of 8080 R&L.)

The FRAME NUMBER and OFFSET fields (present only for absolute segments, A=0 or A=5) specify the placement in MAS of the absolute segment. The range of OFFSET is constrained to be between 0 and 15 inclusive. If a value larger than 15 is desired for OFFSET then an adjustment of the FRAME NUMBER should be done.

The LTL DAT subfield (present only for LTL segments, A=6) specifies the attributes of an LTL segment. It has the following format:

Z’s indicate that these 1-bit fields have- not currently been assigned a function. These bits are required to be zero.

G (Group) is a 1-bit field that, if 1, specifies that the segment is a member of a group, and should be loaded as a part of the group.

BSM (Big Segment Maximum Length) is a 1-bit field that, if 1, specifies that the maximum segment length is exactly 64K. In the case the MAXIMUM SEGMENT LENGTH must contain zero.

The MAXIMUM SEGMENT LENGTH subfield (present only for LTL segments, A=6) specifies the maximum length in bytes of the LTL segment. (The purpose of this field is to provide information to a loader as to reserve memory space as much as possible up to the value in this field.) This value must be greater than or equal to the value in the SEGMENT LENGTH field. The MAXIMUM SEGMENT LENGTH field is only big enough to hold numbers from 0 to 64K-l inclusive. The BSM attribute bit in the LTL DAT field (see above) must be used to give the segment a MAXIMUM length of 64K.

The GROUP OFFSET subfield (present only for LTL segments, A=6) gives the offset of the first byte of the segment relative to the base of the parent group. It must be zero if the G bit is 0. This value will be used by the loader to determine the location relative to the group base of the data records belonging to the segment.

SEGMENT LENGTH

The SEGMENT LENGTH field gives the length of the segment in bytes. The length may be zero: if so, LINK-86 (unlike LINK-80) will not delete the segment from the module. The SEGMENT LENGTH field is only big enough to hold numbers from 0 to 64K-1 inclusive. The B attribute bit in the ACBP field (see above) must be used to give the segment a length of 64K.

SEGMENT NAME INDEX

The Segment Name is a name the programmer or translator assigns to the segment. Examples: CODE, DATA, TAXDATA, MODULENAME_CODE, STACK. This field provides the Segment Name, by indexing into the list of names provided by the LNAMES Record(s).

CLASS NAME INDEX

The Class Name is a name the programmer or translator can assign to a segment. (If none is assigned, the name is null, and has length 0.) The purpose of Class Names is to allow the programmer to define a "handle" by which several LSEG's. may be referred to (e.g. at LOCATE-time) by a single reference. Examples: RED, WHITE, BLUE; ROM, FASTRAM, DISPLAYRAM. This field provides the Class Name, by indexing into the list of names provided by the LNAMES Record(s).

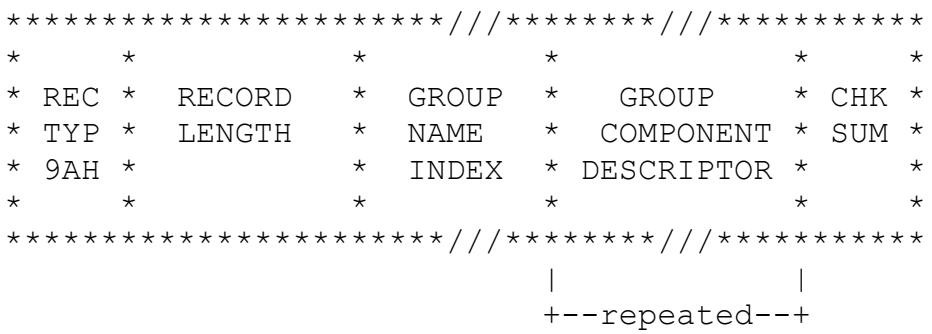
OVERLAY NAME INDEX

The Overlay Name is a name the translator and/or LINK-86, at the programmer's behest, apply to a segment. The Overlay Name, like the Class Name, may be null. This field provides the Overlay Name, by indexing into the list of names provided by the LNAMES Record(s).

> **(Note)** The "Complete Name" of a segment is a 3-component entity comprising a Segment Name, a Class Name and an Overlay Name. (The latter 2 components may be null.) **(End of Note)**

GROUP DEFINITION RECORD

(GRPDEF)



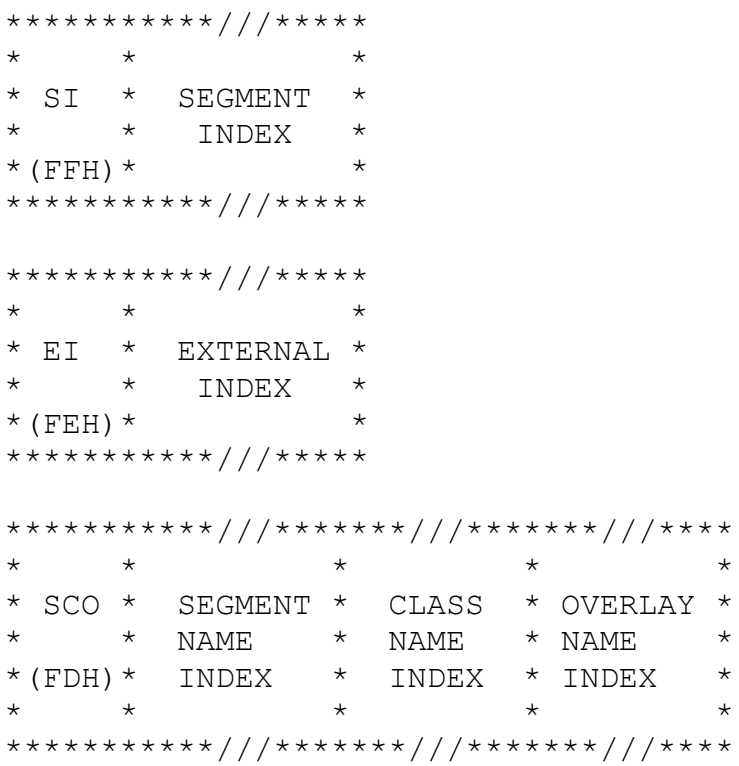
GROUP NAME INDEX

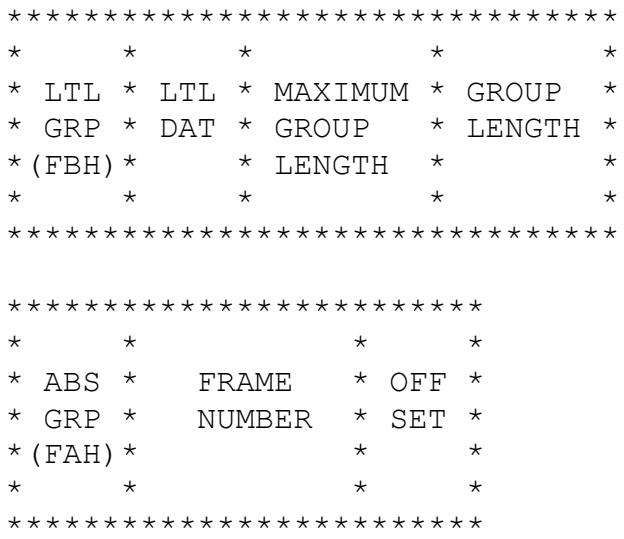
The Group Name is a name by which a collection of 1 or more LSEG's may be referenced. The important property of such a group is that. when the LSEG's are eventually fixed in MAS, there must exist some FRAME which contains (or “covers”) every LSEG of the group. If this is not the case, LOCATE-86 will issue a warning message.

The GROUP NAME INDEX field provides the Group Name, by indexing into the list of names provided by the LNAMES Record(s).

GROUP COMPONENT DESCRIPTOR

Each GROUP COMPONENT DESCRIPTOR has 1 of the following formats:





These 5 kinds of DESCRIPTOR's are now discussed:

If the first byte of the DESCRIPTOR contains 0FFH, then the DESCRIPTOR contains 1 more field, which is a SEGMENT INDEX that selects the LSEG described by a preceding SEGDEF record.

If the first byte of the descriptor contains 0FEH, then the DESCRIPTOR contains 1 more field, which is an EXTERNAL INDEX that selects the LSEG that is (eventually) found to contain the specified External Name.

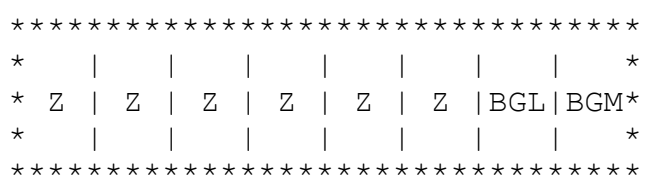
> **(Note)** If the definition of the External Index is (eventually) found to be physical instead of logical (i.e., the External is defined with respect to a PSEG rather than an LSEG), then an error in group specification has occurred. **(End of note)**

If the first byte of the DESCRIPTOR contains 0FDH, then the DESCRIPTOR contains 3 more fields, which are Name Index fields, which determine one or more Segment Name(s), Class Name{s), and Overlay Name(s), respectively. This DESCRIPTOR allows a translator or programmer to include in a group, one or more LSEG's from separate translations (for which SEGMENT INDEX's cannot be known).

A Name Index with value zero carries special significance: it specifies all Names. (Note: Name Indices with zero value may not occur in other record types.)

If the first byte of the DESCRIPTOR contains 0FBH. then the DESCRIPTOR contains 3 more fields. which are the LTL OAT field. The maximum length of the group, and the length of the group. This descriptor, if present. must precede all other descriptors in the record. There may be at most one descriptor of this type in a GRPDEF record. There may not be any absolute component in the group. A segment cannot be in two such groups.

The LTL DATA field has the following format:



Z's indicate that these 1-bit fields have not currently been assigned a function. These bits are required to be zero.

BGL (Big Group Length) is a 1-bit subfield that, if 1, specifies that the Group length is exactly 64K. In this case the GROUP LENGTH subfield must contain zero.

BGM (Big Group Maximum Length) is a 1-bit subfield that, if 1, specifies that the maximum group length is exactly 64K. In this case the MAXIMUM GROUP LENGTH subfield must contain zero.

The GROUP LENGTH subfield specifies the length of the group that has been determined after the Group is "located", and the segments in the group are put in contiguous memory area. All fixups have been performed relative to the base of the Group.

The MAXIMUM GROUP LENGTH subfield specifies the maximum length of the group that has been determined after the Group is "located", using the maximum lengths of the segment components.

If the first byte of the DESCRIPTOR contains 0FAH, then the DESCRIPTOR contains the address of the Group. Once a Group has been LOCATEd, it has an address chosen by LOCATE-86, relative to which all fixups have been performed. If fixups relative to the Group base are required after LOCATE-86 has assigned an address to the Group then the FRAME NUMBER should be used as the base. The address of the Group is also available for debugging systems such as ICE. If a Group has been assigned an address by LOCATE-86 then it is absolute and this descriptor must precede all other descriptors in the record. There may be at most one descriptor of this type in a GRPDEF record.

(Examples) Assume that an LNAMES record exists such that the names "DATA", "RAM", "MYPROG", "CODE", "" (null), "STACK", "CONST" and "MEMORY" are selected by Name Index values of 1, 2, 3, 4, 5, 6, 7 and 8, respectively.

The Descriptor with 4 fields: [0FDH, 3, 1, 1] specifies the LSEG with Segment Name "MYPROG", Class Name "DATA", and no (or "null", or "unspecified") Overlay Name

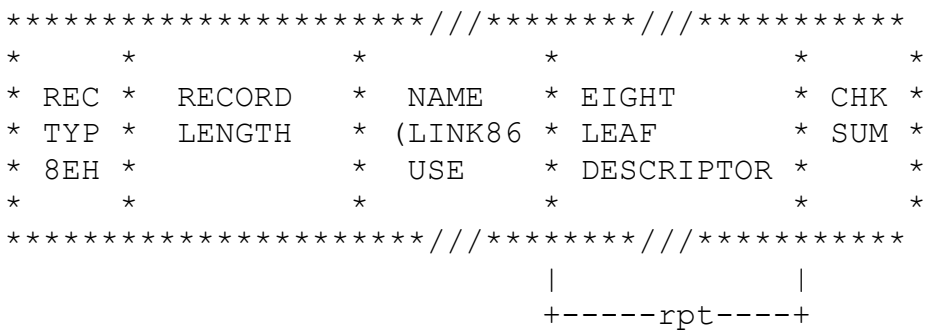
The Descriptor with fields: [0FDH, 3, 1, 5] specifies the LSEG with Segment Name “MYPROG”, Class Name “DATA”, and no (or “null” or “unspecified”) Overlay Name.

The Descriptor with fields: (0FDH, 3, 1, 0] specifies any and all LSEG's with Segment Name “MYPROG” and Class Name “DATA”, regardless of their Overlay Name(s).

The PLM-86 compiler will be able to inform LOCATE-86 of the “Small” assumptions by emitting 2 GRPDEF (Group Definition) Records: one contains the single descriptor [0FDH, 4, 4, 5], the other contains the descriptors [0FDH, 1, 1, 5], [0FDH, 6, 6, 5], [0FDH, 7, 7, 5], and [0FDH, 8, 8, 5]. **(End of Examples)**

TYPE DEFINITION RECORD

(TYPDEF)



This record provides the description of the type of an object or objects presumably named by one or more names provided in PUBDEF, EXTDEF, BLKDEF, DEBSYM and/or LOCSYM records. The type is described as a Branch, which consists of a sequence of Leaves. The types supported, and the corresponding branches, are provided in an appendix.

As many "EIGHT LEAF DESCRIPTOR" fields as necessary, are used to describe a branch. (Every such field except the last in the record describes eight leaves; the last such field describes from one to eight leaves.)

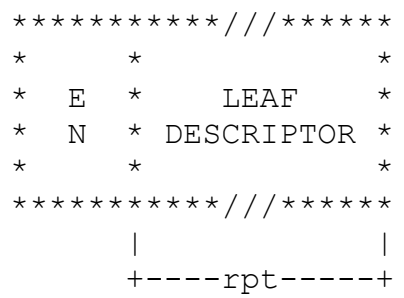
TYPE INDEX values 1 through 32767, which are contained in other record types to associate object types with object names, are defined implicitly by the sequence in which TYPDEF records appear in the object file.

NAME (LINK86 USE)

Use of this field is reserved for LINK-86. Translators should place a single byte containing 0 in it (which is the representation of a name of length zero).

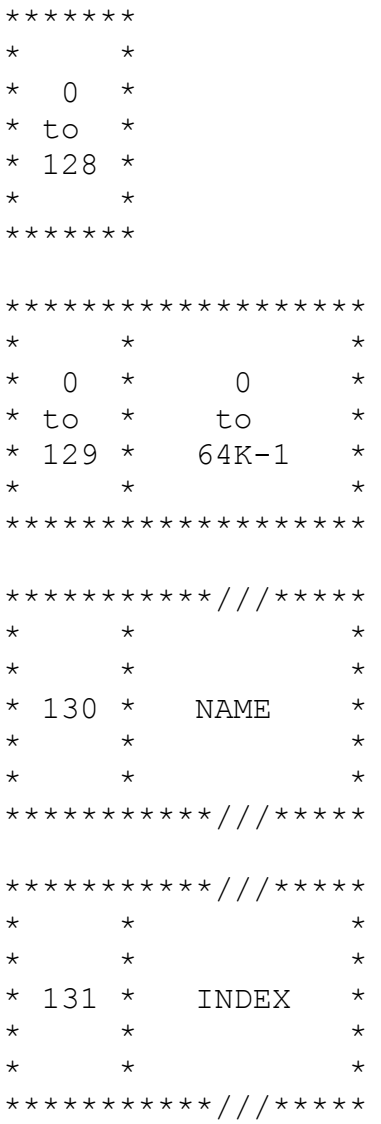
EIGHT LEAF DESCRIPTOR

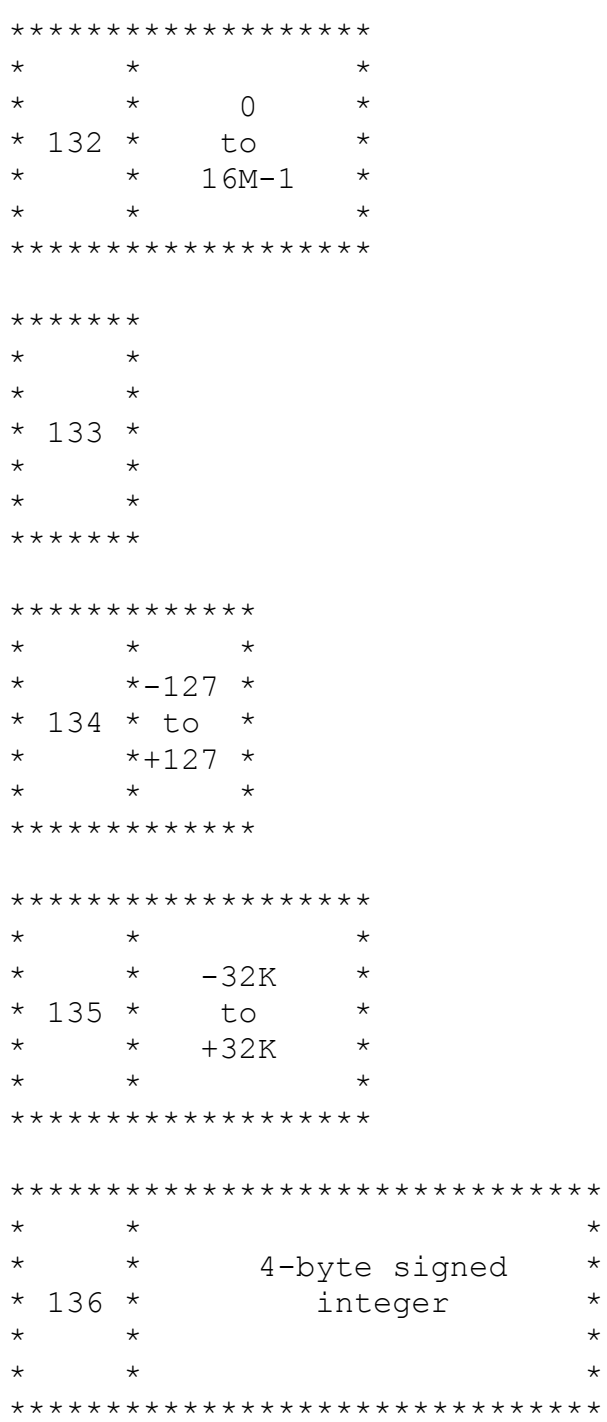
This field can describe up to eight Leaves. If; more than eight, Leaves are to be represented, the field may be repeated as necessary. Unless the last leaf is a Repeat Leaf (see below), the Branch is deemed to end in an indefinite sequence of easy null leaves. This field has the following format:



The EN field is a byte : the 8 bits, left to right, indicate if the following 8 Leaves (left to right) are Easy (bit=0) or Nice (bit=l).

The LEAF DESCRIPTOR field, which occurs between 1 and 8 times, has one of the following formats:





The single byte, containing a value between 0 and 128 represents a Numeric Leaf or Null Leaf. If the value is 128, it represents a Null Leaf. If the value is less than 128, it represents a Numeric Leaf with indicated integer number.

The second form, with a leading byte containing 129, represents a Numeric Leaf. The number is contained in the following 2 bytes.

The third form, with a leading byte containing 130, represents a String Leaf. The field following the leading byte represents the string, in OMF’s standard representation.

The fourth form, with a leading byte containing 131, represents an Index Leaf. The field following the leading byte represents an Index, which is a number between 0 and 32K-1, in OMF's standard representation. Recursively defined types are allowed.

The fifth form, with a leading byte containing 132, represents a Numeric Leaf. The number is contained in the fol1owing 3 bytes.

The sixth form, a single byte of 133, is a Repeat Leaf. A Repeat Laf can only occur as the last leaf of a Branch. If the last leaf of a branch is a Repeat Leaf then the previous leaf is considered to repeat indefinitely. Otherwise the Branch is considered to end in an indefinitely long sequence of easy Null leaves.

The seventh form, with a leading byte containing 134, represents a Signed Numeric Leaf. The number is contained in the following byte, which will be signed extended if necessary.

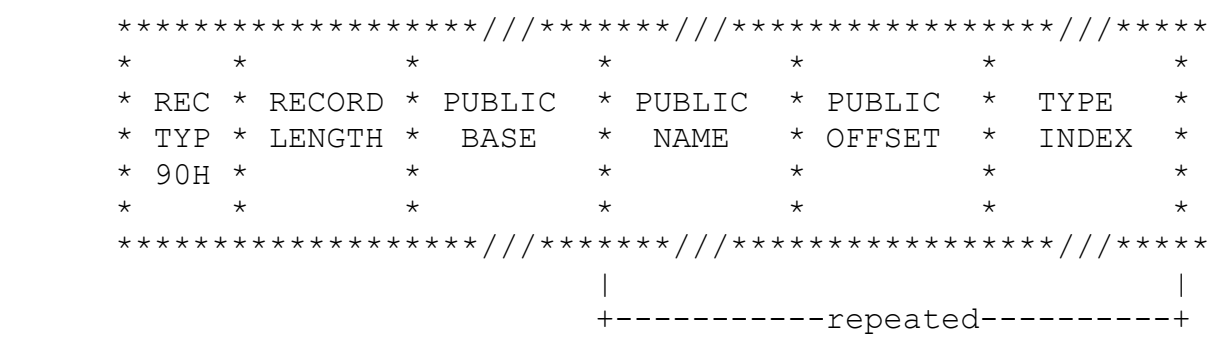
The eighth form, with a leading byte containing 135, represents a Signed Numeric Leaf. The number is contained in the following 2 bytes, signed extended if necessary.

The ninth form, with a leading byte containing 136, represents a Signed Numeric Leaf. The number is contained in the following 4 bytes, signed extended if necessary.

SYMBOL DEFINITION RECORDS

PUBLIC NAMES DEFINITION RECORD

(PUBDEF)



This record provides a list of 1 or more PUBLIC NAME's; for each one, 3 datums are provided: (1) a base value for the name, (2) the offset value of the name, and (3) the type of entity represented by the name.

PUBLIC BASE

The PUBLIC BASE has the following format:

The GROUP INDEX field has a format given earlier, and provides a number between 0 and 32767 inclusive. A non-zero GROUP INDEX “associates” a group with the public symbol, and is used as described on page 15, case (F2c). A zero GROUP INDEX indicates that there is no associated group.

The SEGMENT INDEX field has a format given earlier, and provides a number between 0 and 32767 inclusive.

A non-zero SEGMENT INDEX selects an LSEG, in which case the location of each public symbol defined in the record is taken as a non-negative displacement (given by a PUBLIC OFFSET field) from the first byte of the selected LSEG, and the FRAME NUMBER field must be absent.

A SEGMENT INDEX of 0 (legal only if GROUP INDEX is also 0) means that the location of each public symbol defined in the record is taken is a displacement from the base of the FRAME defined by the value in the FRAME NUMBER field.

(Informal Discussion) The FRAME NUMBER is present if both the SEGMENT INDEX and GROUP INDEX are zero.

A non-zero GROUP INDEX selects some group; this group is taken as the "frame of reference" for references to all public symbols defined in this record, e.g., LINK-86 and LOCATE-86 will perform the following actions:

(1) Any fixup of the form:

```
TARGET:   EI(P)
FRAME:    TARGET
```

(where "P" is a public symbol in this PUBDEF record) will be converted by LINK-86 to a fixup of the form:

```
TARGET:   SI(L),d
FRAME:    GI(G)
```

where "SI(L)" and "d" are provided by the SEGMENT INDEX and PUBLIC OFFSET fields. (The "normal" action would have the frame specifier in the new fixup be the same as in the old fixup, viz.: FRAME: TARGET.)

(2) When the value of a public symbol, as defined by the SEGMENT INDEX, PUBLIC OFFSET, and (optionally) FRAME NUMBER fields, is converted to a {base, offset} pair, the base part will be taken as the base of the indicated group. (If a non-negative 16-bit offset cannot then complete the definition of the public symbol's value, an error will occur.)

A GROUP INDEX of zero selects no group. LINK-86 will not alter the FRAME specification of fixups referencing the symbol, and LOCATE-86 will take, as the base part of the absolute value of the public symbol, the canonic frame of the segment (either LSEG or PSEG) determined by the SEGMENT INDEX field. **(End of Informal Discussion)**

PUBLIC NAME

The PUBLIC NAME field gives the name of the object whose location in MAS is to be made available to other modules. The name must contain 1 or more characters.

(Note) R&L's only constraint upon the characters in names is that they lie within the range 20H (space) through 7EH" (~ tilde) inclusive. Other characters may be used, but may produce awkward results when output to listing devices, etc.

However, translators may proscribe the admissible character set more strictly. **(End of Note)**

PUBLIC OFFSET

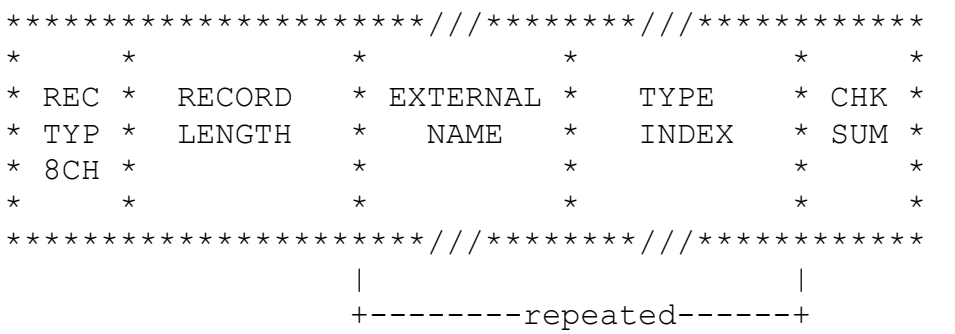
The PUBLIC OFFSET field is a l6-bit value, which is either the offset of the Public Symbol with respect to an LSEG (if SEGMENT INDEX > 0), or the offset of the Public Symbol with respect to the specified FRAME (if SEGMENT INDEX = 0).

TYPE INDEX

The TYPE INDEX field identifies a single preceding TYPDEF (Type Definition) Record containing a descriptor for the type of entity represented by the Public Symbol.

EXTERNAL NAMES DEFINITION RECORD

(EXTDEF)



This Record provides a list of external names, and for each such name, the type of object it represents. LINK-86 will assign to each External Name the value provided by an identical Public Name (if such a name is found), provided that the two names name objects of the same type.

EXTERNAL NAME

This field provides the name, which must have non-zero length, of an external object.

Inclusion of a Name in an External Names Record is an implicit request that the object file be linked to a module containing the same name declared as a Public Symbol. This request obtains whether or not the External Name is actually referenced within some FIXUPP Record in the module.

The ordering of EXTDEF Records within a module, together with the ordering of External Names within each EXTDEF Record, induces an ordering on the set of all External Names requested by the module. Thus, External Names are considered to be numbered: 1, 2, 3, 4, ... These numbers are used as "External Indices" in the TARGET DATUM and/or FRAME DATUM fields of FIXUPP Records, in order to refer to a particular External Name. The format of an External Index has been given earlier.

> **(Caution)** 8086 External Names are numbered positively: 1,2,3, ... This is a change from 8080 External Names, which were numbered starting from zero: 0,1,2,... The reason is to conform with other 8086 Indices (Segment Index, Type Index, etc.) which use 0 as a default value with special meaning. **(End of Caution)**

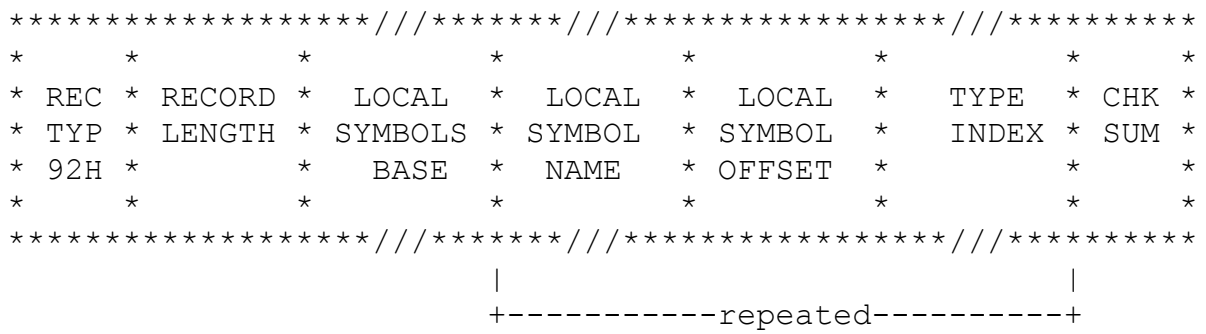
External indices may not be forward referring. That is to say, an external definition record defining the k'th object

must precede any record referring to that object with index k.

TYPE INDEX
This field identifies a single preceding TYPDEF (Type Definition) Record containing a descriptor for the type of object named by the External Symbol.

LOCAL SYMBOLS RECORD

(LOCSYM)



This record provides information about symbols that were used in the source program input to the translator which produced the module. The purpose of this information is to aid ICE and other debugging programs.

The information provided by the LOCSYM record is processed but not used by the R&L products.

The symbols in the record were originally defined in a source module of name given by the most recently preceding T-MODULE HEADER record.

LOCAL SYMBOLS BASE

The LOCAL SYMBOLS BASE has the following format:

The LOCAL SYMBOLS BASE provides two things: (1) it gives a “referent” value (location in MAS), with respect to which the value (location in MAS) of every symbol in the record will be defined by giving, for each symbol in the record, a non-negative offset; and (2) it gives an indication to LOCATE-86 as to how the final (20-bit) values of the symbols should be decomposed into {base,offset} pairs.

The referent value is given by the SEGMENT INDEX or by the FRAME NUMBER. If the SEGMENT INDEX field contains a number greater than 0, then the referent value is the location of the canonic frame of the LSEG specified by the SEGMENT INDEX. (There must be no FRAME NUMBER field in this case.) If both the GROUP INDEX field and the SEGMENT

INDEX field contain zero, then the next field is a FRAME NUMBER: in this case, the referent value is the location of the first byte of the specified frame.

If the GROUP INDEX is zero, the base will be the canonic frame of the LSEG specified by the SEGMENT INDEX (if non-zero), or by the FRAME NUMBER (if SEGMENT INDEX field contains zero). If the GROUP INDEX is non-zero, the base will be the canonic frame of the Group specified by the GROUP INDEX. (If the value of a symbol cannot be described with respect to such a base, then LOCATE-86 will give a warning.)

> **(Note)** When GROUP INDEX is > 0, then one must also have SEGMENT INDEX > 0. **(End of note)**

LOCAL SYMBOL NAME

This field providers the name of the symbol.

LOCAL SYMBOL OFFSET

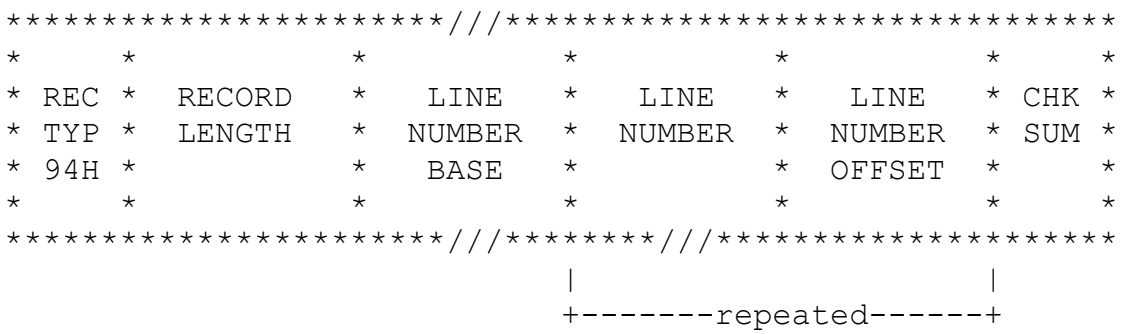
The LOCAL SYMBOL OFFSET is a l6-bit value, which is either the offset of the Local Symbol with respect to an LSEG (if SEGMENT INDEX > 0), or the offset of the Local Symbol with respect to the specified FAME (if SEGMENT INDEX = 0).

TYPE INDEX

The TYPE INDEX field identifies a single preceding TYPDEF Record containing a descriptor for the type of entity represented by the Local Symbol.

LINE NUMBERS RECORD

(LINNUM)



This record provides the means by which a translator may pass to a debugger program, the correspondence between a line number in source code and the corresponding translated code.

Since several independent source modules, with independent line numbering, may be linked to form a single module, a full identification of a source text line must include both its number, and also the name of the original containing module. The latter identification is provided by the previous T-MODULE HEADER Record.

LINE NUMBER BASE

The LINE NUMBER BASE has the following format:

The LINE NUMBER BASE has the same format and interpretation as the LOCAL SYMBOLS BASE described for the LOCSYM record. The SEGMENT INDEX and (if present) the FRAME NUMBER fields determine the location of the first byte of code corresponding to some source line number. This location may be physical (SEGMENT INDEX is 0) or logical (SEGMENT INDEX is non-zero). The value of the GROUP INDEX field, if non-zero, informs LOCATE-86 what base-part to use for describing the final, 20-bit location of the code line. An example shows the use of a non-zero Group Index: A translator knows that the code segment it is compiling is but one LSEG of many in a Group, and thus references to pieces of the code segment are fixed up under the assumption that the appropriate segment register contains the location of the base of the group. At debug

time, the user may tell ICE-86 to "GO TO LINE NUMBER 22 OF MODULE MODNAME". ICE-86 may respond by executing a long jump to the appropriate location. This long jump will set the CS register; it is important that the CS register be set in accordance with the assumptions made while translating the code. This is the purpose of the GROUP INDEX field.

LINE NUMBER

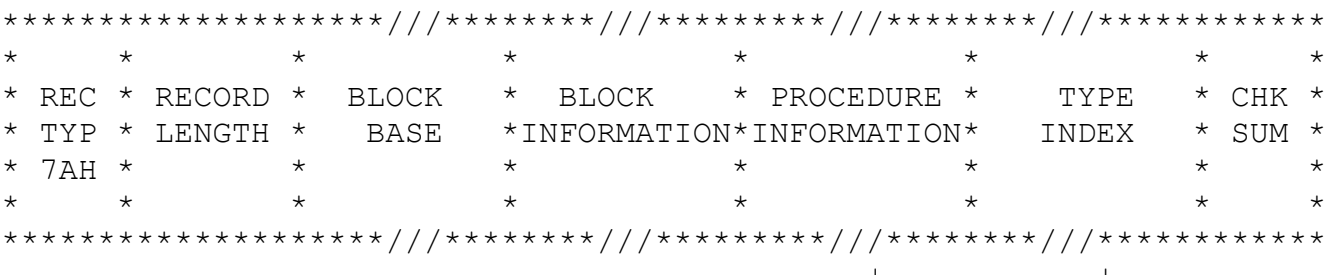
A line number between 0 and 32767, inclusive, is provided in binary by this field. The high order bit is reserved for future use and must be zero.

LINE NUMBER OFFSET

The LINE NUMBER OFFSET field is a 16-bit value, which is either the offset of the line number with respect to an LSEG (if SEGMENT INDEX > 0), or the offset of the line number with respect to the specified FRAME (if SEGMENT INDEX = 0).

BLOCK DEFINITION RECORD

BLKDEF



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

The information provided by the BLKDEF record is processed but not used by the R&L products.

The block in the record was originally defined in a source module of name given by the most recently preceding THEADR record.

BLOCK INDEX values, used in the DEBSYM record, are defined implicitly by the sequence of BLKDEF records in the T-MODULE.

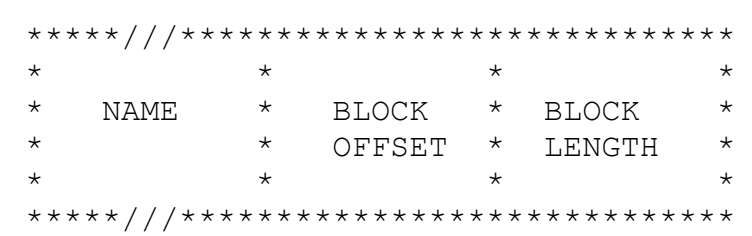
BLOCK BASE

The BLOCK BASE has the following format:

The BLOCK BASE has the same format and interpretation as the LOCAL SYMBOLS BASE described for the LOCSYM record.

BLOCK INFORMATION

The BLOCK INFORMATION block has the following format:



NAME

This field contains the name of the block If the record describes an unnamed block in the source code (e.g., a DO block with no label in PL/M) the NAME will be of length 0.

BLOCK OFFSET

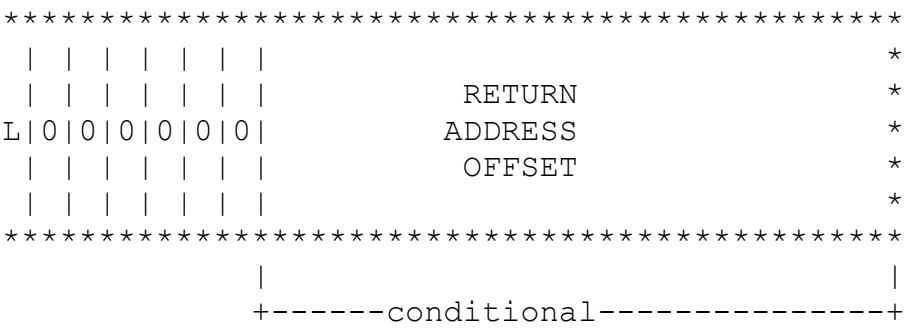
The BLOCK OFFSET is a 16-bit value which is the offset of the 1st byte of the block with respect to the referent value specified by the BLOCK BASE.

BLOCK LENGTH

This field gives the length of the block in bytes.

PROCEDURE INFORMATION

The PROCEDURE INFORMATION block has the following format:



The P (Procedure) bit, if 1, indicates that the BLKDEF record was generated from a procedure in the source. The RETURN ADDRESS OFFSET is present only if P=1.

The L (LONG) bit tells whether the return address is a 4-byte value (CS and IP) or a 2-byte value (IP only).

L=0 -> 2-byte return address
L=1 -> 4-byte return address

If P=0 the L bit has no meaning and is required to be 0.

The RETURN ADDRESS OFFSET, a 16-bit value, is the byte offset (from BP) of the procedure's return address in the procedure's activation record on the stack.

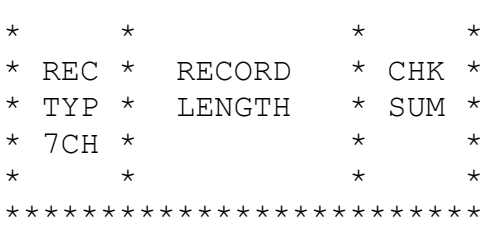
TYPE INDEX

The TYPE INDEX field identifies a single preceding TYPDEF record containing the type descriptor for the procedure or block name. This field is present only if the NAME is non-zero.

> (**Note)** Symbols defined in BLKDEF records should not be repeated in DEBSUM records. **(End of Note)**

BLOCK END RECORD

BLKEND

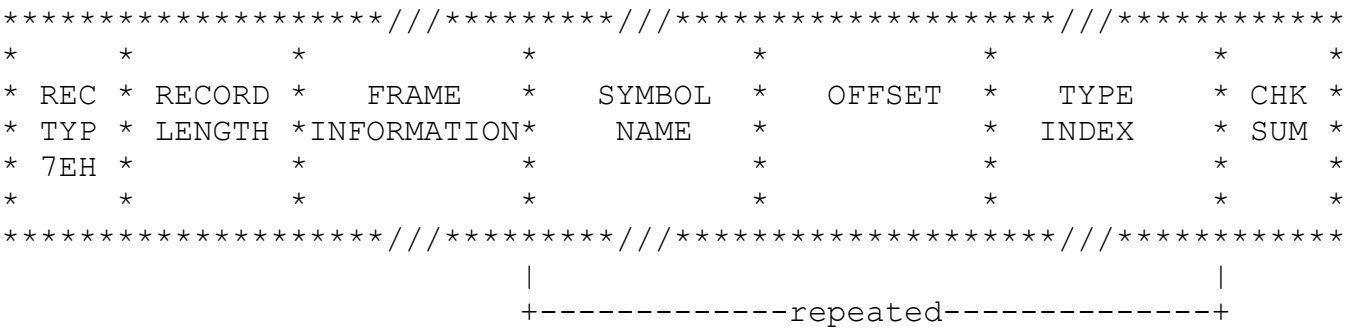


This record, together with the BLKDEF record, provides information about the scope of variables in the source program. Each BLKDEF record must be followed by a BLKEND record. The order of the BLKDEF, debug symbol records, and BLKENDs should reflect the order of declaration in the source module.

> **(Note)** For translators whose variables don't have scope (e.g. ASM86) the ordering of the records need not reflect the order of declaration in the source module. **(End of Note)**

DEBUG SYMBOLS RECORD

(DEBSYM)



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

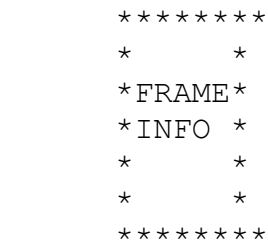
The information in this record is processed but not used by the R&L products.

The symbols in the record were originally defined in a source module of name given by the most recently preceding T-MODULE HEADER record.

The scope of the symbols in the record is defined to be the block of the most recently preceding BLKDEF whose extent has not yet been closed by a BLKEND. If no such BLKDEF exists the symbols are global to the source module of name given by the most recently preceding THEADR.

FRAME INFORMATION

This field gives information about the frame of the symbols defined in the record. It’s format is as follows:



The FRAME INFO byte has the following format:

The B (Based) bit, if 1, means that the location in MAS defined by the FRAME INFORMATION and OFFSET fields contains a value that is the address of a symbol.

The L (Long) bit tells the length of this value.

L=0 -> 2 bytes
L=1 -> 4 bytes

If L=0 the frame part of the symbol address is defined to be the frame given by the FRAME INFORMATION field.

If B=0, the location defined by the FRAME INFORMATION and OFFSET fields is the location of the symbol. In this case the L bit has no meaning and is required to be 0.

The FRAME METHOD field defines what kind of data is in the DATUM field.

If FRAME METHOD=0, the DATUM has the format:

The interpretation of the DATUM fields in the above format is identical to the interpretation of the LOCAL SYMBOLS BASE in the LOCSYM record.

If FRAME METHOD=1, the DATUM has the format:

If FRAME METHOD=2, the DATUM has the format:

FRAME METHODs of 3 to 7 are illegal.

The FRAME METHOD field also specifies what kind of information is in the OFFSET field (see below).

SYMBOL NAME

This field provides the name of the symbol.

OFFSET

The OFFSET field contains a 16-bit value which is interpreted as follows:

If FRAME METHOD is 0 then this field is the offset with respect to the FRAME NUMBER or SEGMENT specified by the DATUM of the FRAME INFORMATION field.

If FRAME METHOD=1 then this field is the byte offset from the external symbol specified by the DATUM of the FRAME INFORMATION field.

If FRAME METHOD=2 then this field is the byte offset (from BP) in the activation record of the block specified by the DATUM of the FRAME INFORMATION field.

TYPE INDEX

The TYPE INDEX field identifies a single preceding TYPDEF record containing a descriptor for the type of entity represented by the symbol.

> **(Note on LOCSYMs)** A DEBSYM record whose FRAME INFO field is 0 is exactly equivalent to a LOCSYM record. **(End of Note on LOCSYMs)**

DATA RECORDS

RELOCATABLE ENUMERATED DATA RECORD

(REDATA)

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. For this reason the record may also be called Load-Time Locatable (LTL) Enumerated Data Record.

The data provided in this record may belong to any LSEG or Group or it may be assigned absolute 8086 memory addresses and be divorced from all LSEG/Group information. The data in this record is subject to modification by FIXUPP records, if any, which follow.

This record may be generated by translators or (8086 based) LINK-86 to produce loadable modules, and will be converted to PEDATA record by the LOCATE-86 program.

DATA RECORD BASE

The DATA RECORD BASE has the following format:

The DATA RECORD BASE specifies the base relative to which the final address of the data record may be defined. It has the same format and interpretation as the LOCAL SYMBOL BASE described for the LOCSYM record.

DATA RECORD OFFSET

This field specifies an offset of the first byte of the DAT field either with respect to an LSEG (if SEGMENT INDEX > 0) or with respect to the specified FRAME (if SEGMENT INDEX = 0). Successive data bytes in the DAT field occupy successively higher locations of memory.

DAT

If one or more FIXUPP records follow then this field provides up to 1024 consecutive bytes of load-time locatable or absolute data. Otherwise, the repeated field is constrained only by the RECORD LENGTH field.

> **(Note on data record size)** All data bytes in a data record must be within the frame specified by the data record. This is true for all 6 data record types (REDATA, RIDATA, PEDATA, PIDATA, LEDATA, LIDATA). **(End of Note on data record size)**

RELOCATABLE ITERATED DATA RECORD

(RIDATA)

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. For this reason the record may also be called Load-Time Locatable (LTL) Iterated Data Record.

The data provided in this record may belong to any LSEG or Group or it may be assigned absolute 8086 memory addresses and be divorced from all LSEG/Group information. The data in this record is subject to modification by FIXUPP records, if any, which follow.

This record may be generated by translators or (8086 based) LINK-86 to produce load able modules, and will be converted to RIDATA record by the LOCATE-86 program.

DATA RECORD BASE

The DATA RECORD BASE has the following format:

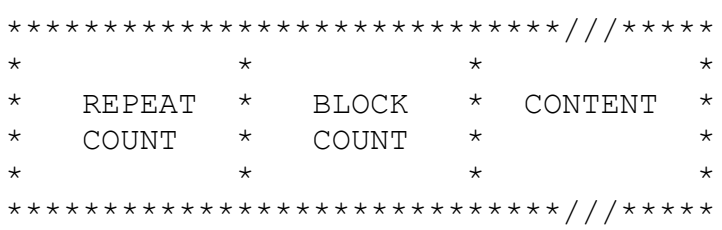
The DATA RECORD BASE specifies the base relative to which the final address of the data record may be defined. It has the same format and interpretation as the LOCAL SYMBOLS BASE described for the LOCSYM record.

DATA RECORD OFFSET

This field specifies an offset of the first byte of the ITERATED DATA BLOCK field either with respect to an LSEG (if SEGMENT INDEX > 0) or with respect to the specified FRAME (if SEGMENT INDEX = 0). Successive data bytes in the ITERATED DATA BLOCK field occupy successively higher locations of memory.

ITERATED DATA BLOCK

This repeated field is a structure specifying the repeated data bytes. It is a structure that has the following format:



REPEAT COUNT

This field specifies the number of times that the CONTENT portion of this ITERATED DATA BLOCK is to be repeated. REPEAT COUNT must be non-zero.

BLOCK COUNT

This field specifies the number of ITERATED DATA BLOCKS that are to be found in the CONTENT portion of this ITERATED DATA BLOCK. If this field has value zero then the CONTENT portion of this ITERATED DATA BLOCK is interpreted as data bytes. If non-zero then the CONTENT portion is interpreted as that number of ITERATED DATA BLOCKs.

CONTENT

This field may be interpreted in one of two ways, depending on the value of the previous BLOCK COUNT field.

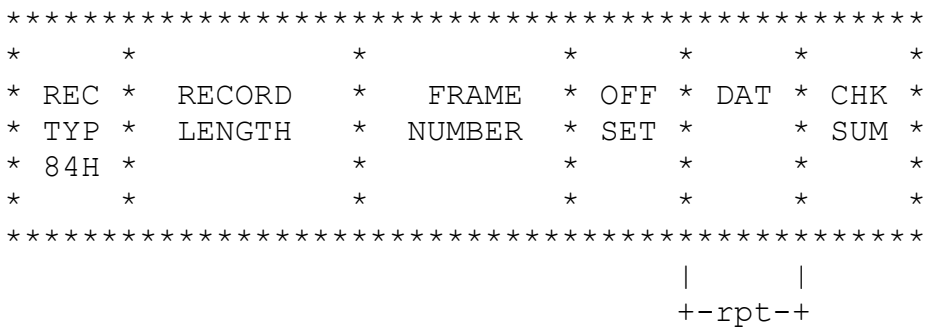
If BLOCK COUNT is zero then this field is a 1 byte count followed by the indicated number of data bytes.

If BLOCK COUNT is non-zero then this field is interpreted as the first byte of another ITERATED DATA BLOCK.

(Note) From the outermost level, the number of nested ITERATED DATA BLOCKS is limited to 17, i.e., the number of levels of recursion is limited to 17. **(End of Note)**

PHYSICAL ENUMERATED DATA RECORD

PEDATA



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 is has been assigned absolute 8086 memory addresses and has been divorced from all LSEG information. The data is subject to modification by FIXUPP records, if any, which follow. If there are FIXUPP records following, then the RECORD LENGTH is constrained to be less than or equal to 1028.

This record may be generated by translators to produce a loadable absolute data record and will be also generated by LOCATE-86.

FRAME NUMBER

This field specifies a Frame Number relative to which the data bytes will be loaded.

OFFSET

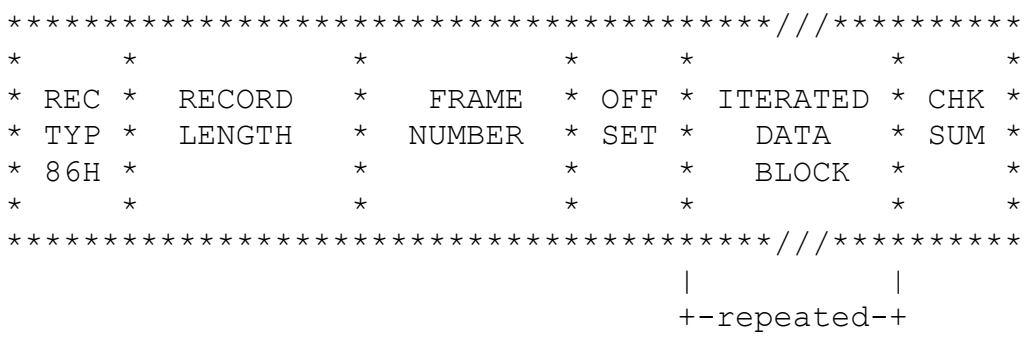
This field specifies an offset relative to the FRAME NUMBER which defines the location of the first data byte of the DAT field. Successive data bytes in the DAT field occupy successively higher locations of memory. The value of OFFSET is constrained to be in the range 0 to 15 inclusive. If an OFFSET value greater than 15 is desired then an adjustment of the FRAME NUMBER should be done.

DAT

If one or more FIXUPP records follow then this field provides up to 1024 consecutive bytes of an 8086 memory image. Otherwise, the repeated field is constrained only by the RECORD LENGTH field.

PHYSICAL ITERATED DATA RECORD

PIDATA



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 are repeated data to be used to initialize a memory image. The data belongs to the “unnamed absolute segment” in that it has been assigned absolute 8086 memory addresses and has been divorced from all LSEG information. The data is subject to modification by following FIXUPP records if any. If there are FIXUPP records then the ITERATED DATA BLOCK length is constrained to be less than 1025.

This record may be generated by translators to produce a loadable absolute data record and will be also generated by LOCATE-86.

FRAME NUMBER

This field specifies a frame number relative to which the data bytes will be loaded.

OFFSET

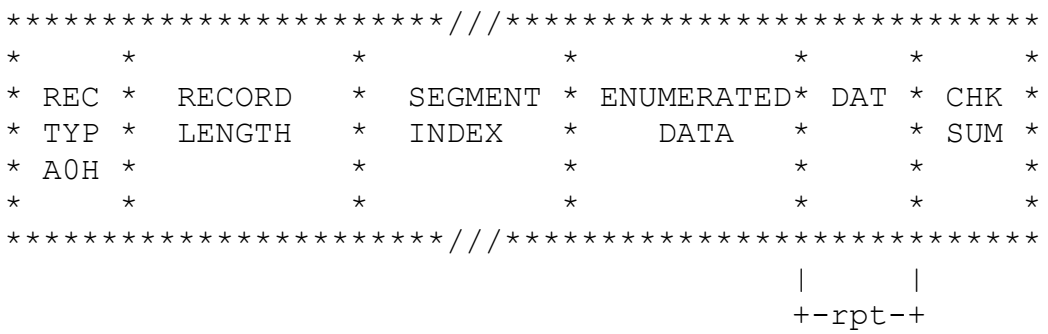
This field specifies an offset relative to the FRAME NUMBER which defines the location of the first data byte in the ITERATED DATA BLOCK. Successive data bytes in the ITERATED DATA BLOCK occupy successively higher locations of memory. The range of OFFSET is constrained to be between 0 and 15 inclusive. If a value larger than 15 is desired for OFFSET then an adjustment of FRAME NUMBER should be done.

ITERATED DATA BLOCK

Same as for RIDATA record.

LOGICAL ENUMERATED DATA RECORD

(LEDATA)



This record provides contiguous data from which a portion of an 8086 memory image may eventually be constructed. The data will probably NOT he loaded directly by an 8086 loader as it must be further processed by the 8086 R&L products.

The data provided in this record may belong to any LSEG. The BASE portion of the address in the case of an absolute segment will be found in the SEGMENT DEFINITION RECORD specified by the SEGMENT INDEX. If the SEGMENT INDEX specifies a segment whose alignment attribute is not absolute then the data provided by this record is relocatable.

This record may be converted to a REDATA RECORD by the (8086 based) LINK-86 program and will be converted to a PEDATA RECORD by the LOCATE-86 program.

SEGMENT INDEX

This field must be non-zero and specifies an index relative to the SEGMENT DEFINITION RECORDS found previous to the LEDATA RECORD. The SEGMENT DEFINITION RECORD may specify that the data is absolute as one of the attributes of the segment. In this case a Frame Number is provided in the SEGDEF record. Absolute data must be able to be placed into LEDATA RECORDs so that grouping of relocatable LSEG's with absolute LSEG's can be achieved.

ENUMERATED DATA OFFSET

This field specifies an offset that is relative to the base of the LSEG that is specified by the SEGMENT INDEX and defines the relative location of the first byte of the DAT field. Successive data bytes in the OAT field occupy successively higher locations of memory. If the SEGMENT INDEX specified an absolute LSEG then the offset is relative to the Frame Number provided in the corresponding SEGDEF RECORD.

DAT

This field provides up to 1024 consecutive bytes of relocatable or absolute data.

LOGICAL ITERATED DATA RECORD

(LIDATA)

This record provides contiguous data, from which a portion of an 8086 memory image may eventually be constructed. The data will probably NOT be loaded directly by an 8086 loader as it must be further processed by the 8086 R&L products.

The data in this record may belong to any LSEG. The BASE portion of the address in the case of named absolute data, will be found in the SEGDEF record specified hy the SEGMENT INDEX. If the SEGMENT INDEX specifies an LSEG other than an absolute LSEG then the data provided by this record is relocatable.

This record may be converted to a RIDATA RECORD by the (8086 based) LINK-86 program and will be converted to a PIDATA RECORD by the LOCATE-86 program.

SEGMENT INDEX

This field must be non-zero and specifies an index relative to the SEGDEF records found previous to the LIDATA RECORD. The SEGDEF record may specify that the data is absolute as one of the attributes of the LSEG. In this case a Frame Number is provided in the SEGDEF record. The LIDATA RECORD is required to allow grouping of relocatable LSEG's with absolute LSEG's.

ITERATED DATA OFFSET

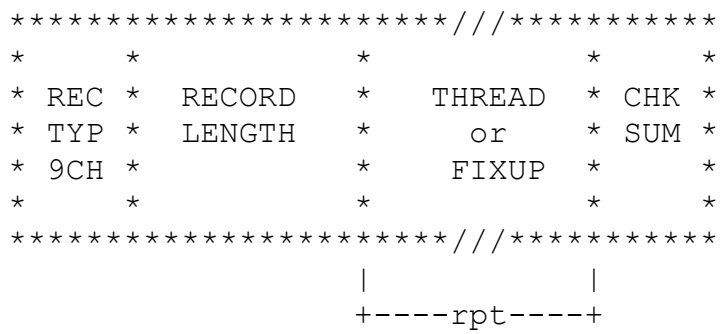
This field specifies an offset that is relative to the base of the LSEG that is specified by the SEGMENT INDEX and defines the relative location of the first byte in the ITERATED DATA BLOCK. Successive data bytes in the ITERATED DATA BLOCK occupy successively higher locations of memory. If the SEGMENT INDEX specified an absolute LSEG then the offset is relative to the Frame Number provided in the corresponding SEGDEF RECORD.

ITERATED DATA BLOCK

Same as for RIDATA record.

FIXUP RECORD

(FIXUPP)

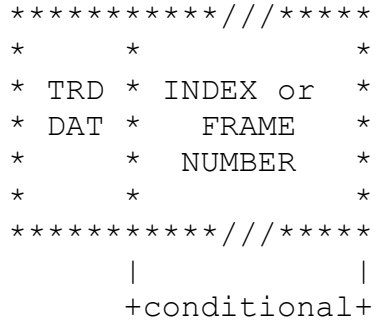


This record specifies 0 or more fixups. Each fixup requests a modification (fixup) to a LOCATION within a previous DATA record. Each fixup is specified by a FIXUP field that specifies 4 data: a location, a mode, a target and a frame. The frame and the target may be specified totally within the FIXUP field, or may be specific by reference to a preceding THREAD field.

A THREAD field specifies a default target or frame that may subsequently be referred to in identifying a target or a frame. Eight threads are provided; four for frame specification and four for target specification. Once a target or frame has been specified by a THREAD, it may be referred to by following FIXUP fields (in the same or following FIXUPP records), until another THREAD field with the same type (TARGET or FRAME) and thread number (0 - 3) appears (in the same or another FIXUPP record).

THREAD

THREAD is a field with the following format:



The TRD DAT (ThReaD DATa) subfield is a byte with this internal structure:

The 'Z' is a one bit subfield, currently without any defined function, that is required to contain 0.

The 'D' subfield is one bit that specifies what type of thread is being specified. If 0=0 then a target thread is being defined and if D=1 then a frame thread is being defined.

METHOD is a 3 bit subfield containing a number between 0 and 3 (D=0) or a number between 0 and 6, (D=1).

If D=0, then METHOD = (0, 1, 2, 3, 4, 5, 6, 7) mod 4, where the 0, ..., 7 indicate methods T0, ..., T7 of specifying a target. Thus, METHOD indicates what kind of Index or Frame Number is required to specify the target, without indicating if the target will be specified in a primary or secondary way.

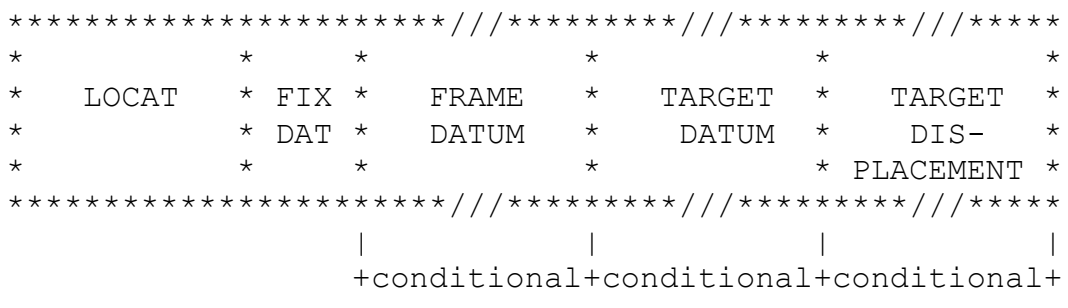
If D=1, then METHOD = 0, 1, 2, 3, 4, 5, 6 corresponding to methods F0, ..., F6 of specifying a frame. Here, METHOD indicates what kind (if any) of Index or Frame Number is required to specify the frame.

THRED is a number between 0 and 3, and associates a “thread number” to the frame or target defined by the THREAD field.

INDEX or FRAME NUMBER contains a Segment Index, Group Index, External Index, or Frame Number depending on the specification in the METHOD subfield. This subfield will not be present if F4 or F5 or F6 are specified by METHOD.

FIXUP

FIXUP is a field with the following format:



LOCAT is a byte pair with the following format:

M is a one bit subfield that specifies the mode of the fixups: self-relative (M=0) or segment relative (M=1).

> (**Note)** Self-relative fixups may NOT be applied to RIDATA, LIDATA, or PIDATA records. **(End of Note)**

S is a one bit subfield that specifies that the length of the TARGET DISPLACEMENT subfield, if present, (see below), in this FIXUP field will be either two bytes (containing a 16-bit non-negative number, S=0) or three bytes (containing a signed 24-bit number in 2's complement form, S=1).

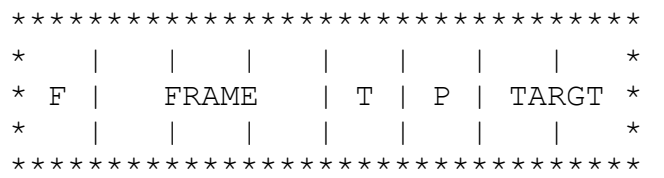
> **(Note)** 3-byte subfields are a possible future extension, and are not currently supported. Thus, S=0 is currently mandatory. **(End of Note)**

LOC is a 3 bit subfield indicating that the byte(s) in the preceding DATA Record to be fixed up are a 'lobyte' (LOC=0), an 'offset' (LOC=1), a 'base' (LOC=2), a 'pointer' (LOC=3), or a 'hibyte' (LOC=4). (Other values in LOC are invalid.)

The DATA RECORD OFFSET is a number between 0 and 1023, inclusive, that gives the relative position of the lowest order byte of LOCATION (the actual bytes being fixed up) within the preceding DATA record. The DATA RECORD OFFSET is relative to the first byte in the data fields in the DATA RECORDs.

> **(Cautionary Note)** If the preceding DATA record is an IDATA record, it is possible for the value of DATA RECORD OFFSET to designate a "location" within a REPEAT COUNT subfield or a BLOCK COUNT subfield of the ITERATED DATA field. Such a reference is deemed an error. LINK-86's and LOCATE-86's action on such a malformed record is undefined, and probably awkward. **(end of Cautionary Note)**

FIX DAT is a byte with the following format:



F is a one bit subfield that specifies whether the frame for this FIXUP is specified by a thread (F=l) or explicitly (F=0).

FRAME is a number interpreted in one of two ways as indicated by the F bit. If F is zero then FRAME is a number between 0 and 6 and corresponds to methods F0, ..., F6 of specifying a FRAME. If F=l then FRAME is a thread number (0-3). It specifies the frame most recently defined by a THREAD field that defined a frame thread with the same thread number. (Note that the THREAD field may appear in the same, or in an earlier FIXUPP record.)

T is a one bit subfield that specifies whether the target specified for this fixup is defined by reference to a thread T=l) or is given explicitly in the FIXUP field (T=0).

P is a one bit subfield that indicates whether the target is specified in a primary way (requires a TARGET DISPLACEMENT, P=0) or specified in a secondary way (requires no TARGET DISPLACEMENT, P=1). Since a target thread does not have a primary/secondary attribute, the P bit is the only field that specifies the primary/secondary attribute of the target specification.

TARGT is interpreted as a two bit subfie1d. When T=0, it provides a number between 0 and 3, corresponding to methods T0, ..., T3 or T4, ..., T7, depending on the value of P (P can be interpreted as the high order bit of T0, ..., T7). When the target is specified by a thread (T=l) then TARGT specifies a thread number (0-3).

FRAME DATUM is the “referent” portion of a frame specification, and is a Segment Index, a Group Index, an External Index, or a Frame Number. The FRAME DATUM subfield is present only when the frame is specified neither by a thread (F=0) nor explicitly by methods F4 or F5 or F6.

TARGET DATUM is the “referent” portion of a target specification. and is a Segment Index, a Group Index, an External Index or a Frame Number. The TARGET DATUM subfield is present only when the target is not specified by a thread (T=0).

TARGET DISPLACEMENT is the 2- or 3-byte displacement required by “primary” ways of specifying TARGETs. This 2- or 3-byte subfie1d is present if P=0.

OVERLAY DEFINITION RECORD

(OVLDEF)

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

OVERLAY NAME

The OVERLAY NAME field provides a name by which a collection of 1 or more LSEG's and/or Groups may be referenced for loading.

The ordering of OVLDEF Records within a module induces an ordering on the set of all Overlays defined in the module. Thus, OVLDEF records are considered to be numbered: 1, 2, 3, 4, ... These numbers are used as “Overlay Indices” in the OVERLAY ATTR field of following OVLDEF records.

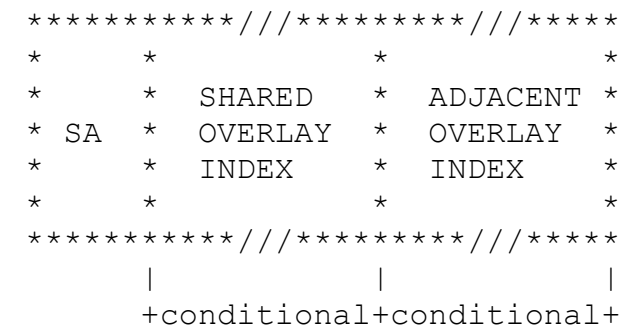
Overlay indices may not be forward referring. That is to say, an overlay definition record defining the k’th overlay must precede any record referring to that overlay with index k.

OVERLAY LOCATION

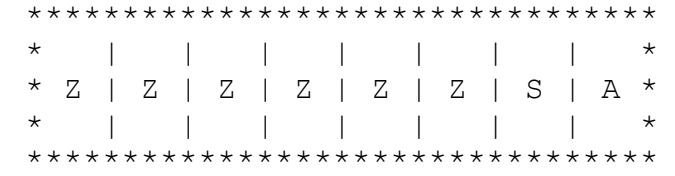
The OVERLAY LOCATION is a 4-byte field which gives the location in bytes relative to the start of the file of the first byte of the records in the overlay.

OVERLAY ATTR

The OVERLAY ATTR field has the following format:



The SA subfield provides in information for memory layout. It has the following format:



Z's indicates that these 1-bit field have not been assigned a function. These bits are required to be zero.

S (shared) is a 1-bit field that, if 1, indicates that the overlay will have to be loaded at the same location as the overlay indicated in the SHARED OVERLAY INDEX field.

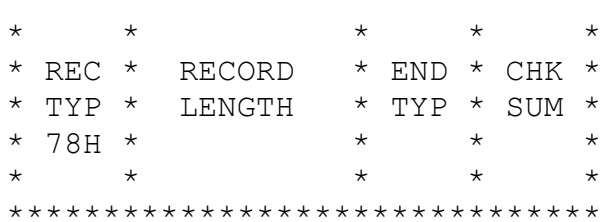
A (adjacent) is a 1-bit field that, if 1, indicates that the overlay will have to be loaded next in memory to the overlay indicated in the ADJACENT OVERLAY INDEX field.

The SHARED OVERLAY INDEX subfield, present if bit S in the SA subfield is 1, points to a previously defined OVLDEF record and indicates that the segments with same segment names and class names and/or the groups with same names in the two overlays must be loaded at the same location.

The ADJACENT OVERLAY INDEX subfield, present if bit A in the SA subfield is 1, points to a previously defined OVLDEF record and indicates that the segments and/or groups in the overlay defined by the current OVLDEF record must be loaded adjacent to the ones with the same names in the indicated overlay.

END RECORD

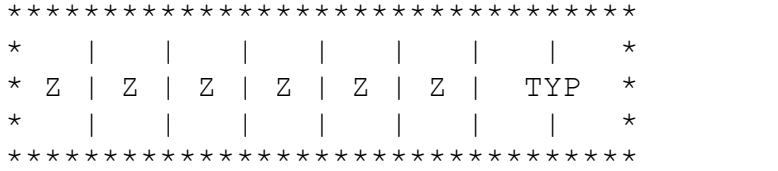
(ENDREC)



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

END TYP

This field specifies the type of the set. It has the following format:



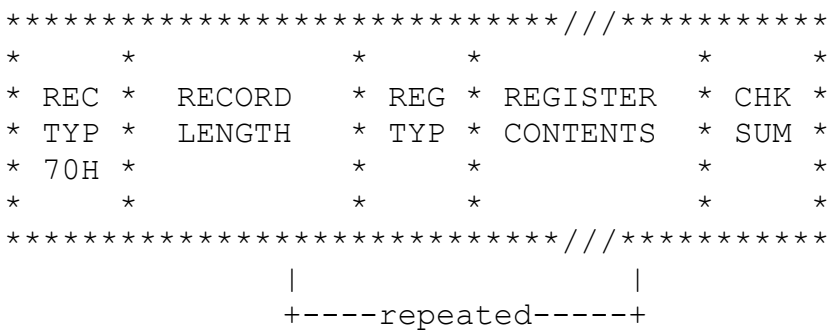
TYP is a two bit subfield that specifies the following types of ends:

TYP	TYPE OF END
0	End of overlay
1	End of block
2	(Illegal)
3	(Illegal)

Z indicates this bit has not currently been assigned a function. These bits are required to be zero.

REGISTER INITIALIZATION RECORD

(REGINT)



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

REG TYP

The REG TYP field provides the register/register-pair name. It also indicates the type of register content specification given in the next field. It has the following format:

Z's are 1-bit subfields which indicate that these bits have not currently been assigned a function. These bits are required to be zero.

REGID is a two bit subfield that specifies the name of the registers/register-pairs as follows:

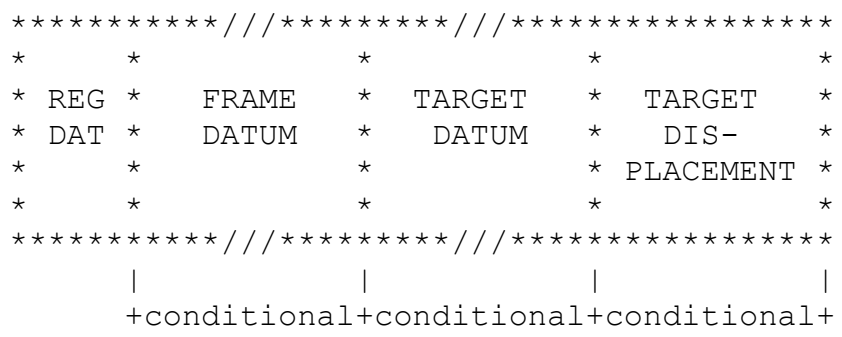
REGID	REGISTER/REGISTER-PAIR
0	CS and IP
1	SS and SP
2	DS
3	ES

L is a one bit field that indicates whether the REGISTER CONTENTS field is to be interpreted as a logical address (L=1) that requires fixing up by LINK-86/LOCATE-86, or as a pair of base and offset specifications (L=0) appropriate for the initialization of the corresponding register/register-pair.

REGISTER CONTENTS

The REGISTER CONTENTS field has either of the following formats:

First form (L=1)



In this case the register contents are specified in exactly the same manner as in the specification of the mapping of a logical address to a physical address as used in the discussion of fixups and the FIXUPP record. The above subfields of the REGISTER CONTENTS field have the same semantics as the FIX DAT, FRAME DATUM, TARGET DATUM, and TARGET DISPLACEMENT fields in the FIXUPP record. Frame method F4 is not allowed.

Second form (L=0)

LINK-86/LOCATE-86 will convert the above REGISTER CONTENTS field into a field having the following format:

The REGISTER BASE field has the following format:

The format and the interpretation of the above REGISTER BASE field is identical to the LOCAL SYMBOLS BASE described in the LOCSYM record.

The REGISTER OFFSET field (present only if REGID <= 1) specifies an offset relative to the Segment (if SEGMENT INDEX > 0) or to the FRAME (if SEGMENT INDEX = 0)

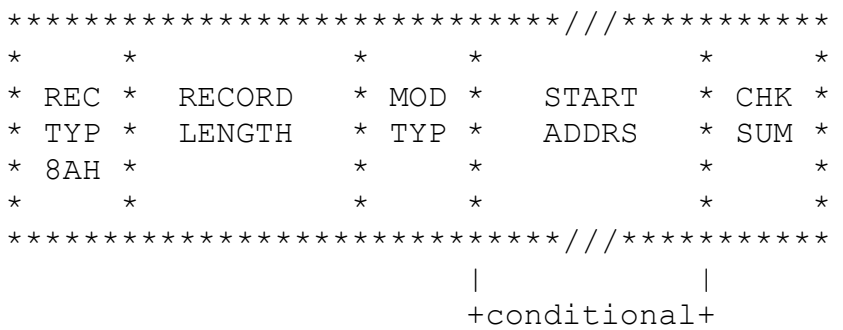
(Note) Once the segments and/or groups are absolutely located (by a loader or LOCATE-86), the base of the object pointed to by the REGISTER BASE field is the appropriate value for the initialization of the corresponding base register. The offset value for the initialization of either the IP register (REGID = 0) or the SP register (REGID = 1) is determined as follows:

If GROUP INDEX = 0, the offset value is given by the value specified in the REGISTER OFFSET field.

If GROUP INDEX > 0, the offset value is the offset relative to the base of the specified group of the location specified by the pair (SEGMENT INDEX, REGISTER OFFSET). **(End of Note)**

MODULE END RECORD

(MODEND)



This record serves two purposes. It denotes the end of a module and indicates whether the module just terminated has a specified entry point for initiation of execution. If the latter is true then the execution address is specified.

MOD TYP

This field specifies the attributes of the module. The bit allocation and associated meanings are as follows:

MATTR is a two bit subfield that specifies the following module attributes:

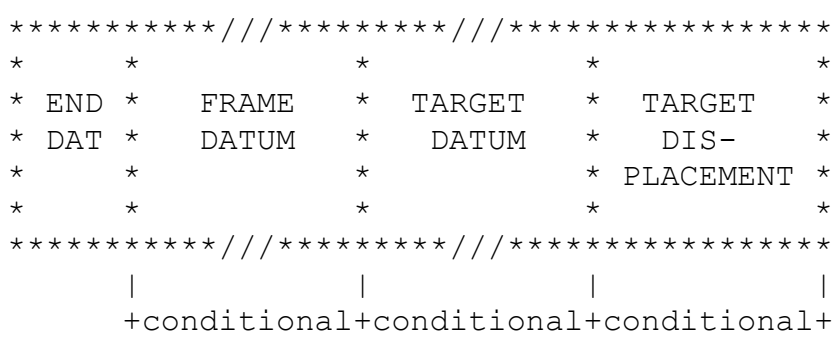
MATTR	MODULE ATTRIBUTE
0	Non-main module with no START ADDRS
1	Non-main module with START ADDRS
2	Main module with no START ADDRS
3	Main module with START ADDRS

L indicates whether the START ADDRS field is to be interpreted as a logical address that requires fixing up by LINK-86/LOCATE-86 (L=1) or as a physical address appropriate for placement into the CS and IP registers of the 8086 (L=0).

Z indicates that this bit has not currently been assigned a function. These bits are required to be zero.

The START ADDRS field (present only if MATTR is 1 or 3) has either of the following formats:

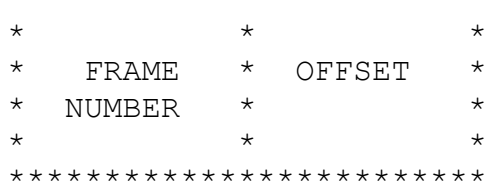
START ADDRS (first form)



The starting address of a module has all the attributes of any other logical reference found in a module. The mapping of logical starting address to a physical starting address is done in exactly the same manner as mapping any other logical address to a physical address as specified in the discussion of fixups and the FIXUPP record. The above subfields of the START ADDRS field have the same semantics as the FIX DAT, FRAME DATUM, TARGET DATUM, and TARGET DISPLACEMENT fields in the FIXUPP record. Only “primary” fixups are allowed. Frame method F4 is not allowed.

START ADDRS (second form)

When the logical address is mapped, by LOCATE-86, to a physical address, the START ADDRS field takes on the following format:



FRAME NUMBER specifies a frame number relative to which the module will begin execution. This value is appropriate for insertion into the CS register for program initiation.

OFFSET specifies an offset relative to the FRAME NUMBER which defines the exact location of the first byte at which to begin execution. This value is appropriate for insertion into the IP register for program initiation.

LIBRARY RECORDS

LIBRARY HEADER RECORD

(LIBHED)

REC TYP A4H	RECORD LENGTH	MODULE COUNT	BLOCK NUMBER	BYTE NUMBER	CHK SUM

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: LIBRARY MODULE NAMES RECORD, LIBRARY MODULE LOCATIONS RECORD, and LIBRARY DICTIONARY RECORD.

MODULE COUNT

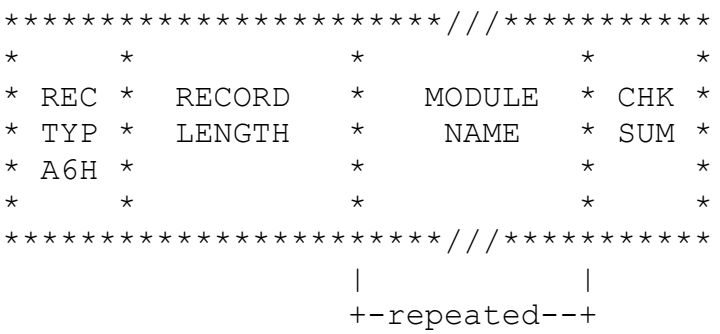
This field indicates how many modules are in the library. It may have any value, including zero.

BLOCK NUMBER, BYTE NUMBER

These fields indicate the relative location of the first byte of the LIBRARY MODULE NAMES RECORD in the library file, using the ISIS-II file format.

LIBRARY MODULE NAMES RECORD

(LIBNAM)



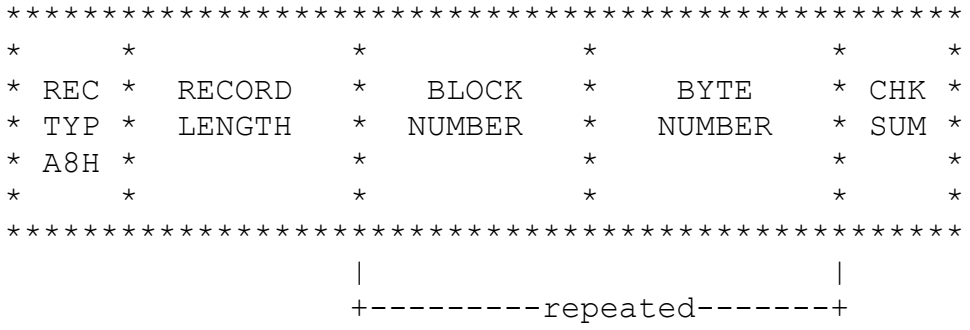
This record gives the names of all the modules in the library. The names are given in the same sequence as the modules appear in the library.

MODULE NAME

The i'th MODULE NAME field in the record contains the module name of the i'th module in the library.

LIBRARY MODULE LOCATIONS RECORD

(LIBLOC)



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.

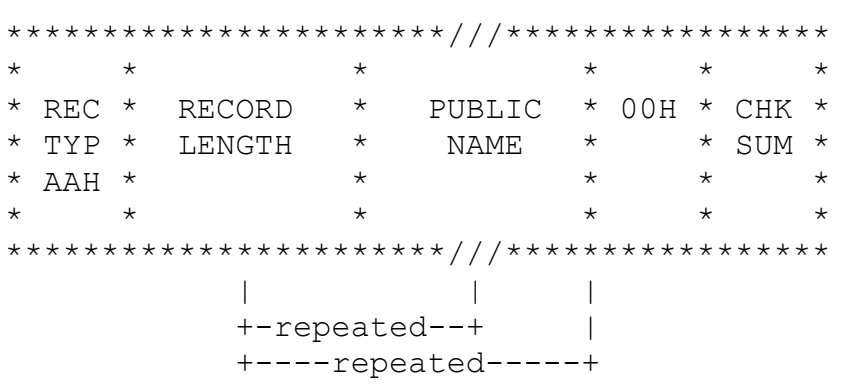
The order of the block-number/byte-number pairs corresponds to the order of the modules within the library.

BLOCK NUMBER, BYTE NUMBER

The i'th pair of fields provides the relative location within the library file of the first byte of the first record of the i'th module within the library, using the ISIS-II file format.

LIBRARY DICTIONARY RECORD

(LIBDEC)



This record gives all the names of public symbols within the library. Since a name must have a non-zero length, the ‘00’ bytes in the format are distinguishable from the PUBLIC NAME fields. Thus, the ‘00’ bytes separate the PUBLIC NAMES into groups; all names in the i’th group are defined in the i’th module of the library.

COMMENT RECORD

(COMENT)

This record allows translators to include commentary information in object text.

COMMENT TYPE

This field indicates the type of comment carried by this record. This allows commentary information to be structured for those processes that wish to selectively act on comments. The format of this field is as follows:

The NP (NOPURGE) bit, if 1, indicates that the COMENT record is not purgeable by object file utility programs which implement the capability of deleting COMENT record.

The NL (NOLIST) bit, if 1, indicates that the text in the COMMENT field is not to be listed in the listing file of object file utility programs which implement the capability of listing object COMENT records.

The COMMENT CLASS field is defined as follows:

0	Language translator comment
1	Intel copyright comment. The NP bit must be set.
2-155	Reserved for Intel use.
156-255	Reserved for users. Intel products will apply no semantics to these values.

COMMENT

This field provides the commentary information.

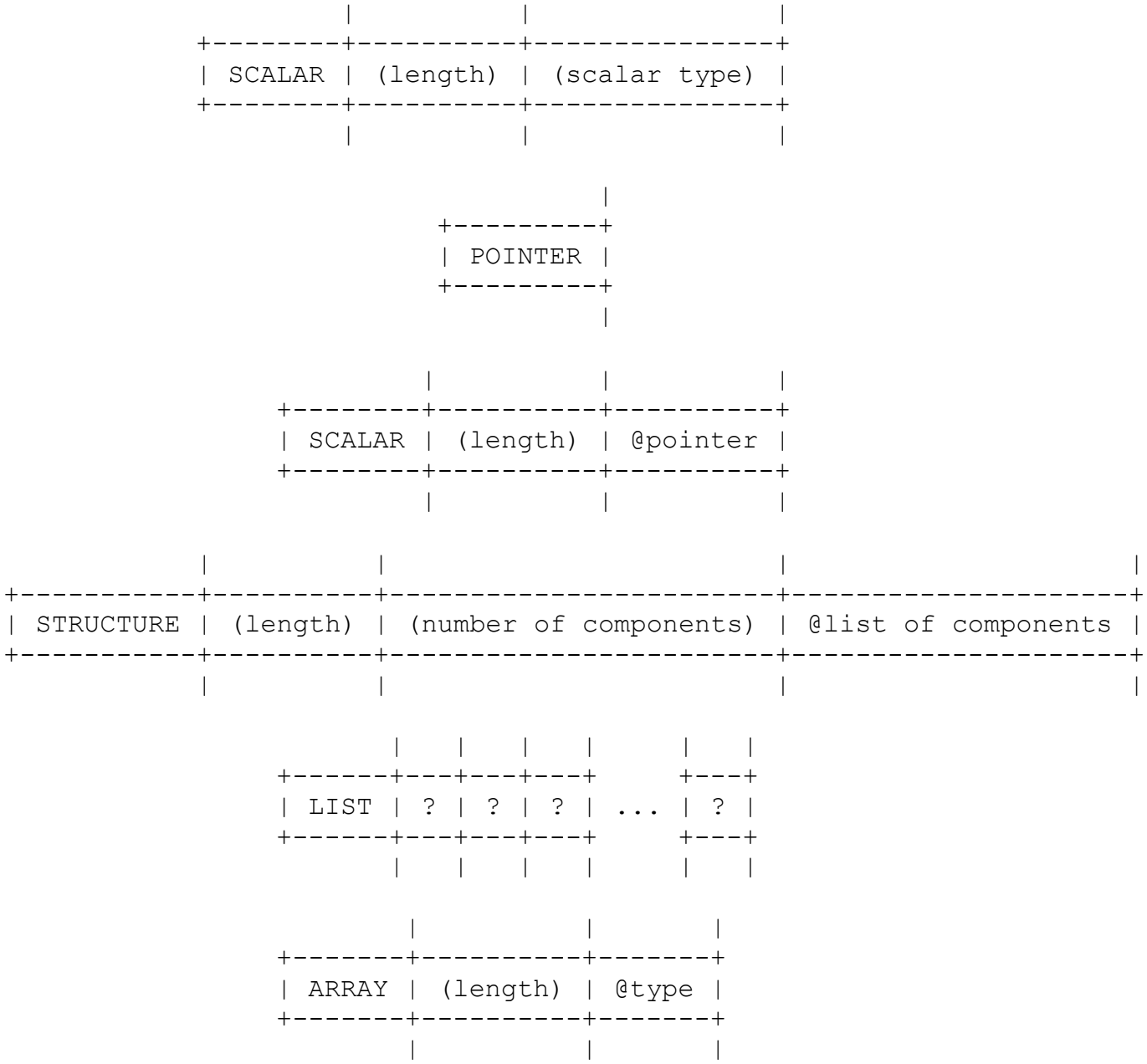
APPENDIX 1 - NUMERIC LIST OF RECORD TYPES

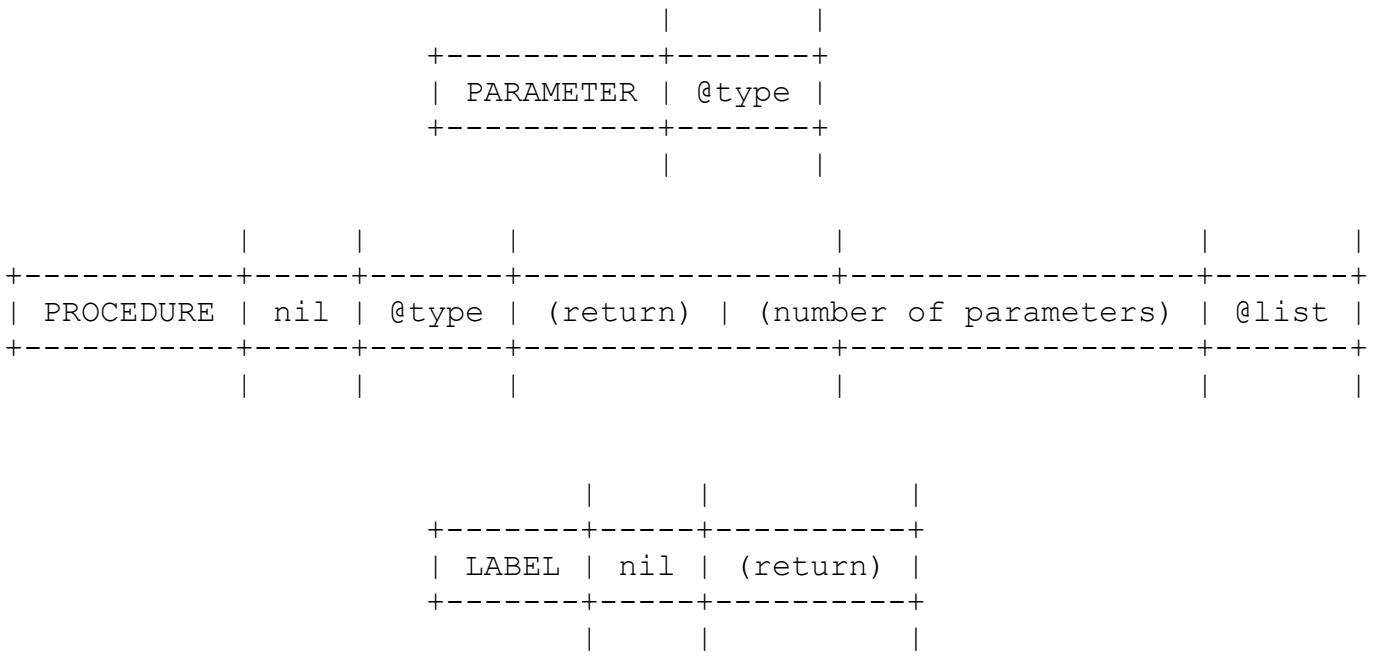
6E RHEADR
70 REGINT
72 REDATA
74 RIDATA
76 OVLDEF
78 ENDREC
7A BLKDEF
7C BLKEND
7E DEBSYM
80 THEADR
82 LHEADR
84 PEDATA
86 PIDATA
88 COMENT
8A MODEND
8C EXTDEF
8E TYPDEF
90 PUBDEF
92 LOCSYM
94 LINNUM
96 LNAMES
98 SEGDEF
9A GRPDEF
9C FIXUPP
9E (none)
A0 LEDATA
A2 LIDATA
A4 LIBHED
A6 LIBNAM
A8 LIBLOC
AA LIBDIC

APPENDIX 2 - TYPE REPRESENTATIONS

The leaves in the following diagrams may be Numeric Leaves without relations, String Leaves, Index Leaves or Null Leaves. Andleaves and Orleaves are not supported at this time.

Types may be defined by branches of the following forms:





where "(scalar type)" can be either UNSIGNED INTEGER, SIGNED INTEGER, or REAL, "(return)" can be either SHORT or LONG (which indicates, in the case of a LABEL, whether a jump to the label should be a "short" jump or a "long" jump, respectively), and the following values are assigned:

99 INTERRUPT	112 (reserved for 1ength)
100 FILE	113 LABEL
101 PACKED	114 LONG
102 UNPACKED	115 SHORT
103 SET	116 PROCEDURE
104 (reserved for 1enqth)	117 PARAMETER
105 CHAMELEON	118 DIMENSION
106 BOOLEAN	119 ARRAY
107 TRUE	120 (reserved for length)
108 FALSE	121 STRUCTURE
109 CHAR	122 POINTER
110 INTEGER	123 SCALAR
111 CONST	124 UNSIGNED INTEGER
	125 SIGNED INTEGER
	126 REAL
	127 LIST

(Note) 1. The above (decimal) values are chosen for the convenience of utility programs such as EDOJ86, and OJED86. All numbers are different (although conceptually there is no reason why REAL and SCALAR, for example, can't be the same number), and are rather 1arqe, so that object module display programs may correctly decide whether to represent a Numeric Leaf as a number or as an identifier, make this choice correctly most of the time, and never give a wrong identifier.

2. For more detailed type descriptions see the translator EPS's (e.g. ASM-86, PLM-86, PASCAL-86, FORTRAN-86). **(end of Note)**

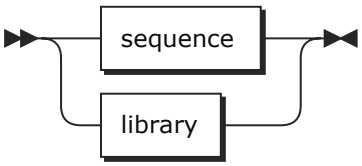
APPENDIX 3 - SYNTAX DIAGRAMS

EBNF

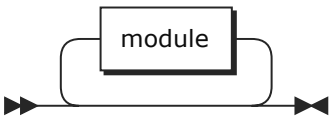
```
object_file ::= (sequence | library)
sequence ::= module*
library ::= LIBHED module* LIBNAM LIBLOC LIBDIC
module ::= (tmod | lmod | rmod | omod)
tmod ::= THEADR sgr_table component* modtail
lmod ::= LHEADR sgr_table data* t_component* modtail
rmod ::= RHEADR sgr_table data* t_component* modtail
omod ::= RHEADR sgor_table o_component* o_modtail
sgr_table ::= seg_grp REGINT?
sgor_table ::= seg_grp OVLDEF* REGINT?
seg_grp ::= LNAMES* SEGDEF* (TYPDEF | EXTDEF | GRPDEF)?
modtail ::= REGINT? MODEND
o_modtail ::= OVLDEF* REGINT? MODEND
o_component ::= data* t_component* ENDREC
t_component ::= THEADR component*
component ::= (data | debug_record)
data ::= (content_def | thread_def | TYPDEF | PUBDEF | EXTDEF)
debug_record ::= (LOCSYM | LINNUM | BLKDEF | BLKEND | DEBSYM)
content_def ::= (LIDATA | LEDATA | PIDATA | PEDATA | RIDATA | REDATA) FIXUPP*
thread_def ::= FIXUPP
```

Diagrams

object_file:

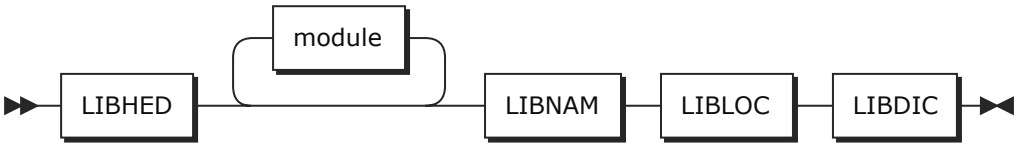


sequence:



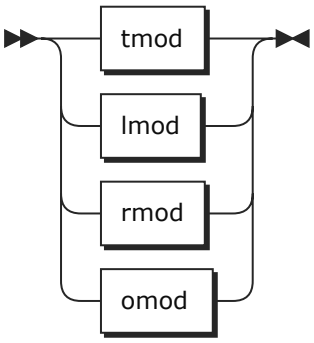
referenced by object file

library:



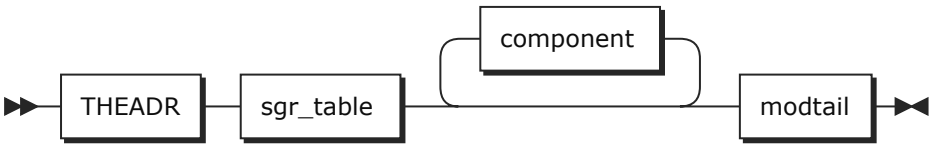
referenced by object file

module:



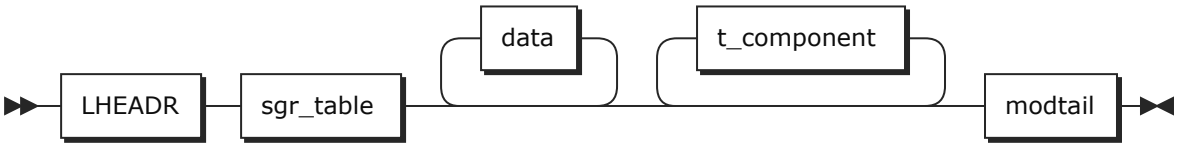
referenced by library, sequence

tmod:



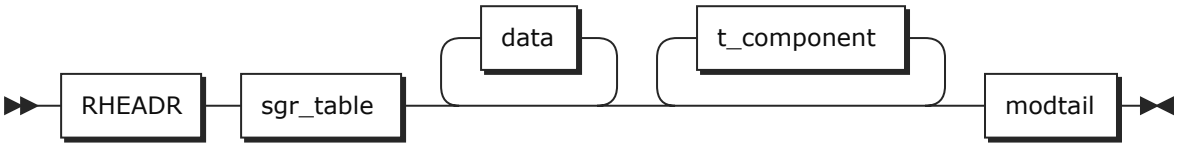
referenced by module

lmod:



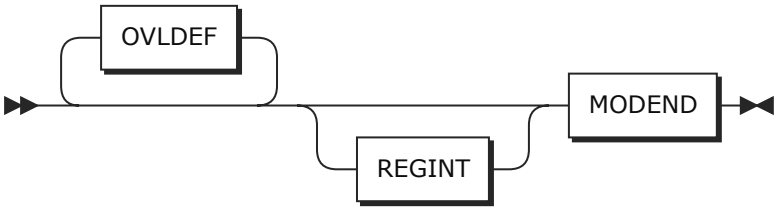
referenced by module

rmod:



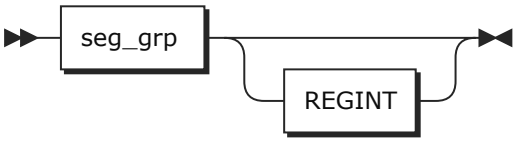
referenced by module

omod:



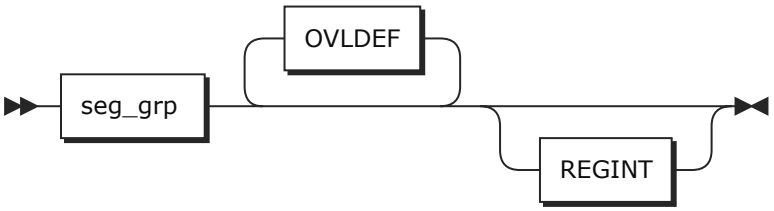
referenced by module

sgr_table:



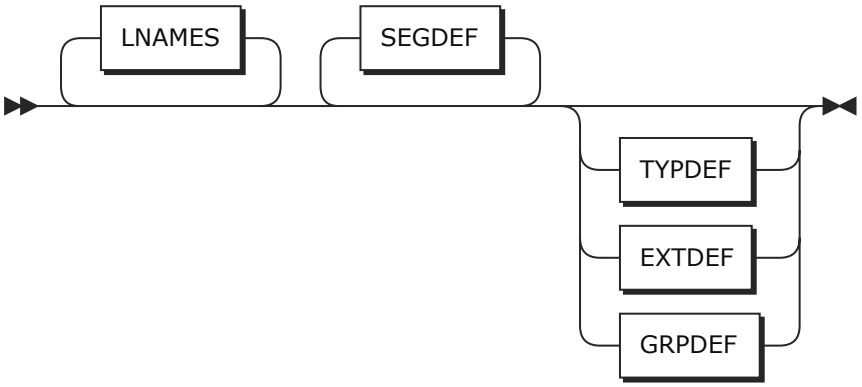
referenced by lmod, rmod, tmod

sgor_table:



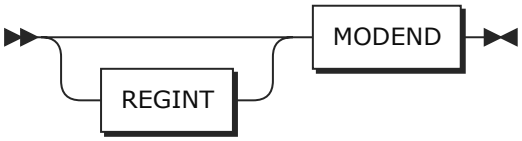
referenced by omod

seq_grp:



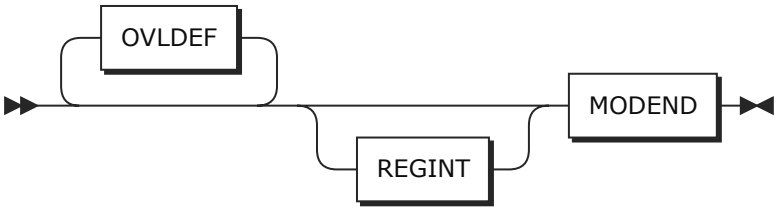
referenced by sgor table, sgr table

modtail:



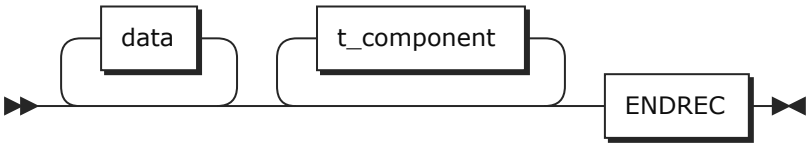
referenced by lmod, rmod, tmod

o_modtail:



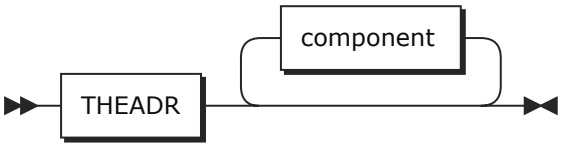
referenced by omod

o_component:



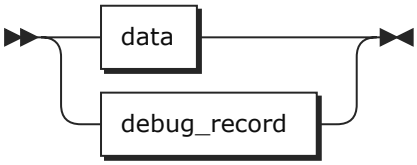
referenced by omod

t_component:



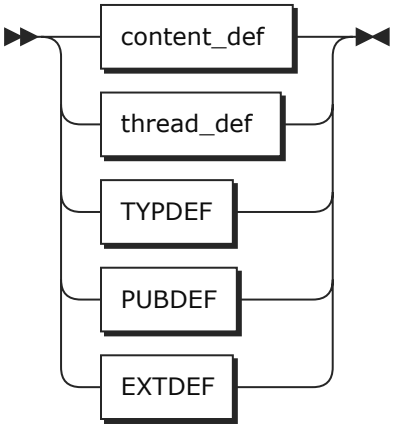
referenced by lmod, o component, rmod

component:



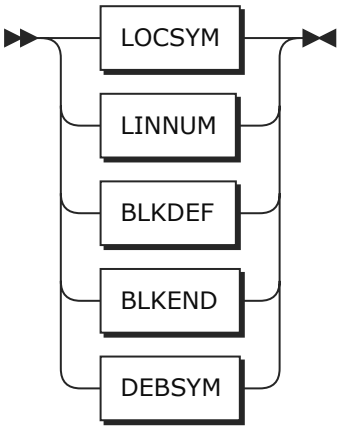
referenced by t component, tmod

data:



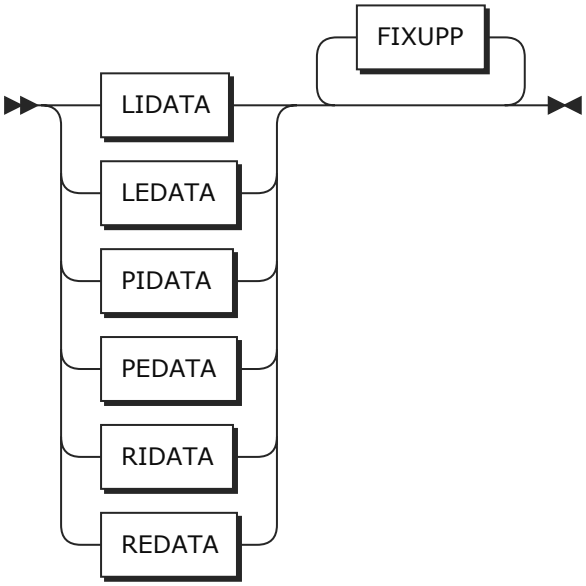
referenced by component, lmod, o component, rmod

debug_record:



referenced by component

content_def:



referenced by data

thread_def:

referenced by data

APPENDIX 4 – EXAMPLES OF FIXUPS

TBD.