IEEE - 695 Object Module Format Specification

Revision 4.1

December 21, 1992

Implementation Defined by:

Microtec Research Incorporated and Hewlett Packard Company

(Page intentionally left blank)

To register to receive updates and errata notices for this specification, please send the portion of the page below the dashed line to:

Hewlett-Packard
Attn: Technical Publications Manager
Logic Product Support Dept.
P.O.Box 2197
Colorado Springs, CO 80901-2197

Please send updates and errata notices for

IEEE - 695 Object Module Format Specification,
Implementation defined by: Microtec Research Inc. and Hewlett-Packard Company,
Revision 4.1, December 21, 1992

to:

Name: _____

Title: _____

Company: _____

Address: _____

City/State/Zip: _____

Phone: _____

(Page intentionally left blank)

1. Introduction

This document† describes the Microtec Research Inc. (MRI)/Hewlett-Packard Company (HP) object module format supporting assemblers, compilers, linkers and debuggers. It is derived from the IEEE format, Standard 695, and includes extensions and limitations necessary to support MRI and HP product requirements. The standard describes both an ASCII and binary version of the format. MRI and HP utilize the more compact binary form. This document should be reviewed in conjunction with the IEEE standard [1].

2. Terminology

The IEEE specification defines a term that is redefined in this document. The term applies to the basic division of an object file which is referred to as a ''command.'' Since this conflicts with the MRI/HP use of command, the basic unit is renamed to be a ''record.'' Object module records are prefixed with a record type byte in the range $E0 through $FF. The term ''library'' is used throughout to mean a single file with more than one relocatable module. The term ''MAU'' is used throughout to mean Minimum Addressable Unit; e.g. a byte (8 bits) on the MC68000 or a word (16 bits) on a Mil-Std-1750A.

2.1 Nomenclature

The following nomenclature is used throughout this document:

- Braces { } surround a required field
- Brackets [ ] surround an optional field
- Dollar Signs ($) precede character representations of hexadecimal numeric values

2.2 Number Format

Numbers are used to define byte counts for fields and to specify numeric parameters. These specifications can have two forms:

- If the value is between 0-127 decimal, the number is $0-$7F.
- If the value is greater than 127 decimal, then the number must be defined by 1 byte of count with the high order bit set ($80) followed by the indicated number of bytes of numeric data with the most significant byte first. The range for the count is usually 0-4 (i.e. $80-$84) and can be 0-8 on some installations. This form is also valid for numbers in the range 0-127.

  Example: $7FFF is encoded as {$82}{$7F}{$FF} (3 bytes). 0 can be encoded as {$00} or {$81}{$00}, 2^^32 can be encoded as {$85}{01}{00}{00}{00}{00}, etc.
- Omitted optional fields in records may be represented by a byte count of zero.

  Example: {$80}
- Numeric fields are represented in the document as {n} and {x}.
- Numeric fields in miscellaneous records are represented as {v}.

---

[1] *IEEE Trial Use Standard for Microprocessor Universal Format for Object Modules, (IEEE Std 695)*, IEEE Technical Committee on Microcomputers and Microprocessors of the IEEE Computer Society, September, 1985.

2.2.1 Negative Numbers

Negative numbers have these characteristics:

- Numbers use $80-$88.
- All readers must handle $80-$84.
- Some processors can use $85 through $88 (see the appendix for your processor).
- Numbers are construed as unsigned except where indicated as signed.
- Signed numbers use signed complement notation; negative numbers must set highest bit of largest representable number (e.g., {$84}{$FF}{$FF}{$FF}{$FF} is -1 for 32-bit and smaller processors).

2.3 Name Format

Name fields are represented in this document by {Id} and consist of 1-byte of count (0-127) followed by the indicated number of ASCII characters. The MRI/HP format extends the IEEE specification to allow the use of any printable ASCII character in a name. Characters are represented as hexadecimal values in the file but are represented as quoted characters in this document for improved readability.

Example:

name ‘‘ABCD’’ = {$04}{$41}{$42}{$43}{$44}

- Name fields in miscellaneous records are represented as {s}.

2.3.1 Long Strings

The IEEE format allows only for printable strings. This implementation allows for non-printable strings.

Path names with network headers, command lines in the environment, foreign characters, and strings of data could exceed 127 characters. An extension byte allows for more than 127 characters. If the reader encounters a DE character, the next one byte is the string length. The one byte length allows strings from 0 to 255 characters. If the reader encounters a DF character, the next two bytes are the string length. The two byte string allows 0 to 65535 characters.

Prefix	Description
$00-$7F	Simple number in the range 0 to 127, or 7-bit ASCII string with length 0 to 127.
$80-$84	Number larger than 127 or negative. 0 to 4 bytes follow. $80 is used as a place holder and means the value was not provided.
$85-$88	Unsigned numbers between 2^^32 and (2^^64)-1, or negative. Not supported in all installations.
$89-$8F	Unused.
$90-$9F	Reserved functions and operators.
$A0-$BF	Function values (arithmetic and logical).
$C0-$DA	Variable letters (null, A-Z).
$DB-$DD	Unused.
$DE-$DF	Extension length. If DE, the next byte is the length of an 8-bit string between 0 and 255 bytes long. If DF, the next two bytes in high-order/low-order format are the length of an 8-bit string between 0 and 65535 bytes long.
$E0-$FA	Record headers.
$FB	Define Context.
$FC-$FF	Unused.

Table 2-1. Initial Bytes of IEEE Elements

2.4 Information Variables

Information variables convey information to a symbolic debugger or linker about various constructs within the program. The information conveyed relates to symbols, section addresses and lengths, starting addresses, and current PC value. These are represented by an alphabetic letter optionally followed by a number:

A*n* The size of a contiguously-mapped portion of a section *n* in MAUs. Multiple assignments of a single A variable indicate successive mappings. Only type B sections may have A-values.

B*n* The physical address of a contiguously-mapped portion of a section *n*. Multiple assignments of a single B variable indicate successive mappings. Only type B sections may have B-values.

F*n* Specifies the size of an addressable memory unit for this section in terms of the MAU for this target. The F variable describes processors (e.g. 8051) that have more than one way of addressing memory. If the value of an ASF record is a positive *n*, it means that *n* addressable units in this section make one MAU for the target. If the value is a negative n, it means n target MAUs make a addressable unit for this section. Only type B or T sections may have F-values.

G Execution starting address.

I*n* Address of public symbol *n*.

L*n* The logical address of a section *n*. Typically, L-values represent the value of the processor's logical address bus. This value does not change within an object file.

When there is no address mapping (i.e., logical addresses are the same as physical addresses) the L variable represents the entire address value.

M*n* The most significant half of a two part logical address of a section *n*. The meaning of an M-value is target-dependent. The M value may be omitted altogether for a particular target. Typically, M-values represent a ''memory space'' number. If not assigned, the M-value for a section equals the M-value of its parent or zero if the parent does not exist or does not assign the value. Only B or T sections may have M-values.

N*n* Address of local symbol *n*.

P*n* The program counter for section *n*; implicitly changes with each LR, LD, or LT that applies to section *n* in the Data part.

R*n* R-values are similar to L-values; they represent the logical address of a section module *n*. Unlike an L-value, an R-value may change within a module.

When there is no address mapping (i.e., logical addresses are the same as physical addresses) the R variable represents the entire address value.

S*n* The size, in MAUs, of a section *n*. Does not change in an object module. A size of 2^^32 bytes is specified as ${85}{$01}{$00}{$00}{$00}{$00}.

W*n* If *n* is 0 through 7, W*n* is the file offset, in bytes, of the *n*'th part of the object file from the beginning of the file. For *n*=8 to 31, the meaning of W*n* is reserved for special uses. Values of W*n* for *n* greater than 31 are available to store values, serve as forward references, etc.

X*n* Address of external symbol *n*.

The number, if present for symbol definitions, identifies which of several variables of the same type is referenced. Therefore, ''I3'' represents public symbol number 3 in the current module. This number is referred to as an ''index'' in the discussion that follows. There are 3 different series of indices: external reference indices, section indices and public name/type/local name indices. Indices must be unique within a module for each series and must be included with all variable specifications except G. Public/local (I/N) type symbol indices between 0 and 31 are reserved for special class symbols. Normal symbol indices begin at 32.

External indices between 0 and 10 are also reserved for special class symbols; normal externals begin at 11. Indices have been defined for the 68000, HD64180, Z80, and 8085. To find the indices that have been defined for your target, see the appropriate appendix in this manual.

Specification of G variables must not include an index. The IEEE standard has been extended to require index values for L, S, and P variables (these are all section indices). The binary encoding for the letters A-Z is $C1-$DA respectively.

2.5 Line Numbers

Object modules can have a significant number of line number records included in typical situations. To minimize the impact upon the size of the object module, the MRI/HP standard defines only one NN record per source file. A line number will be specified by ATN and ASN records only.

2.6 Expressions

Expressions resolve address values at load/locate time. These expressions are coded in the Polish postfix form: ''operand operand operator'' for binary operators, ''operand operator'' for unary operators. Operators are encoded as $A0-$B8 as described in the standard (for example, $A5 is ''+'' , $A6 is ''-'' and $A3 is unary ''-''). An operand can be a number, an information variable, or another expression. Expressions are not explicitly terminated. The record prefix byte ($E0-$FF) for the next record serves as the terminator.

Tables 2-2a and 2-2b show all operator encodings.

Code $B9 is @ESCAPE, rather than the standard's @ISDEF. @ESCAPE is a postfix ''functor'' -- it returns a function as determined by its only argument.

Codes $BA through $BF are expression-bracketing marks. They are described in Section 3.7.6.

Hex	Function	Hex	Function
A0	@F	B0	@AND
A1	@T	B1	@OR
A2	@ABS	B2	@XOR
A3	@NEG	B3	@EXT
A4	@NOT	B4	@INS
A5	+	B5	@ERR
A6	-	B6	@IF
A7	/	B7	@ELSE
A8	*	B8	@END
A9	@MAX	B9	@ESCAPE
AA	@MIN	BA	[††
AB	@MOD	BB	] ††
AC	<	BC	{ ††
AD	>	BD	} ††
AE	=	BE	(††
AF	!= or < >	BF	) ††

Table 2-2a. Function and Operator Encodings †

Encoding	Function
0 @ESCAPE	reserved
1 @ESCAPE	@ISDEF
2 @ESCAPE	@TRANS
3 @ESCAPE	@SPLIT
4 @ESCAPE	@INBLOCK
5 @ESCAPE	@CALL_OPT

Table 2-2b. ESCAPE Function Encodings †

2.6.1 Escape Functions

This section describes the escape functions listed in Table 2-2b.

Defined Variables Function - @ISDEF

@ISDEF takes an expression as its operand. The function returns TRUE if the expression contains no unassigned variables. It returns FALSE otherwise. This expression is encoded differently, although it behaves as described in the *IEEE-695 Trial Use Standard*.

format:

expression @ISDEF

---

† For the meanings of these functions, see the *IEEE Trial Use Standard for Microprocessor Universal Format for Object Modules* (*IEEE Std 695*).

†† The bracket functions shown in Table 2-2a are for illustrative purposes only. Do not confuse these bracket functions with the representational nomenclature for required and optional fields introduced in Section 2.1 and used throughout this document.

Translation Function - @TRANS

The translation function, @TRANS, translates expressions between memory spaces. The operator takes two parameters: an expression to be translated and a value representing the mode of translation (e.g., logical address to physical address). The precise meaning of @TRANS is target dependent; see the appendix for your microprocessor.

format:

variable mode @TRANS

Insert Function - @SPLIT

The insert function, @SPLIT, inserts bit patterns into expressions. This function takes four parameters:

format:

x y z w @SPLIT

where:

x= expression
y= insertion pattern
z = low bit
w = high bit

Thus, the function @SPLIT inserts pattern *y* (truncated to *w*+1-*z* bits) into expression *x* at bit position *z* through *w*. The original bits of *x* at position *z* or higher, are shifted up by (*w*+1-*z*).

For example, in the following:

aaaaaaaaaa 1 8 12 @SPLIT

1 is inserted at bit positions 8 through 12 in the expression aaaaaaaaaa. The original bits at positions 8 and higher are shifted up by (12+1-8=5). The result is:

```
new expression:   aaa00001 aaaaaaaa
bit position:       12   8 7       0
```

In-block Addressing Function - @INBLOCK

The in-block addressing function, @INBLOCK, checks addressing within blocks.

format:

d,s,b @INBLOCK

where:

b=block size
d=destination of jump
s=source of jump

The function returns *d* but optionally reports an error if *d* div *b* does not equal *s* div *b*. The assembler creates the @INBLOCK function. Only the linker reads it and reports any errors.

This operation can be performed by a series of standard IEEE operators but it is cumbersome.

Linker Call Optimization - @CALL_OPT

The @CALL_OPT function provides information to a linker in support of link-time procedure calling optimizations.

format:

parameters opt_code @CALL_OPT

where:

parameters = parameters for the link-time call optimization. The total number of @CALL_OPT parameters will depend on the @CALL_OPT optimization type code.
opt_code = code to specify a particular optimization.

The only currently defined opt_code is 0 for 80960 call optimizations. Code 0 is interpreted as follows:

format:

expr Xn 0 @CALL_OPT

where:

expr = default relocatable expression (used for call instruction)
Xn = an X variable having index n which is to checked for possible .leafproc/.sysproc information
0 = code to specify 80960 call optimization.

The linker will use the information specified in the @CALL_OPT function to generate the appropriate call instruction in the data image part.

2.6.2 Absolute Addresses of Objects

For certain targets, if the bank or memory space number of a static object (symbol, code, starting address, etc.) cannot be determined from context, then the absolute address of the object is represented by an L or R index and an optional offset rather than a single constant.

L*n* offset + or R*n* offset +

2.7 Types

Symbol types supply information to debug and analysis tools to aid in determining the size, organization, and type of program object referenced by the symbol. Tables A-1 and A-2 (Appendix A) describe the symbol classifications supported by MRI/HP language systems and debug tools. Each symbol has an associated type number and/or a mnemonic ‘‘code letter’’ which serves as a shorthand identifier for the type in the object file and elsewhere. The tables indicate the rationalization for using that letter.

2.7.1 Complex Types

Table A-1 identifies the supported high level complex types. These types must be explicitly defined using an IEEE ’TY’ directive (see Section 3.6.3) in order to correctly represent the use of the symbol type in the high level language source code. Table A-1 shows what parameters are used to define the type, where these parameters appear in the IEEE, and the NN and TY records which define the type.

2.7.2 Built-in Types

Table A-2 identifies the implicit or ‘‘built-in’’ types supported by HP/MRI tools. The built-in types represent C, PASCAL, and FORTRAN type definitions for common scalar types (and pointers to common scalar types) which are implicit to the compiler, assembler, linker, and debugger. As for complex types, the type number or mnemonic letter code for built in types implies the size and organization of the program object. The type number also specifies a default type name for use by debug tools in referring to the built-in type.

Built-in types normally do not require additional information other than the type number to completely describe them. Only the number of the built-in type is used in an ATN record describing a symbol having one of the implicit types. It is also the number used in the definitions for more complex types which have elements that are of built-in type. The shorthand notation for implicit types is intended to minimize the size of object modules by providing a short notation for the common subsets of more general types.

If the user (or compiler) chooses to redefine the name of a built-in type, this must be done with a TY record for type T defining a new name for the built-in type and indicating the built-in type as the underlying type. Redefining the name of a built-in type nullifies most of the efficiency gained through use of built-in type codes.

The interpretation of built-in types is processor-dependent. For example, the C type ‘‘int’’ might be 16 bits on one chip and 32 bits on another. Finally, assembler symbols associated with EQUs, DCBs, DCs and DSs are mapped

into types B, H, or L depending on size. Assembler labels not associated with data declarations are mapped as type J. The mapping of C, PASCAL, and FORTRAN scalar types into HP/MRI types is also shown in Table A-2.

The following assumptions relating to typedefs are made by MRI and HP tools:

- Type ‘‘char’’ is assumed to be signed if not explicitly redefined.
- The size assumed for int/unsigned is the stack-push size for a given target (i.e. 68000 = 4 MAUs, Z80 = 2 MAUs, etc.) unless redefined explicitly.
- The size assumed for a pointer is the natural size for the target (i.e. 68000 = 4 MAUs, Z80 = 2 MAUs, etc.) unless explicitly redefined. If two sizes of pointers are possible, NEAR and FAR qualifiers are used to specify which size.

3. Object File Components

An object file is divided into 7 component parts. Each part is a contiguous group of bytes within the file. The component parts may occur in any order within the file with the exception that the Header must occur first and the Module End must occur last. The Header part contains information pointing to the location of the other parts within the file. Therefore, the various file parts do not necessarily have to be read in the order in which they appear. The component parts listed below are described in the following sections:

Header Part

- Module Beginning (MB) - $E0
- Address Descriptor (AD) - $EC
- Assign Pointer to AD Extension Part (ASW0) - $E2D700
- Assign Pointer to Environment Part (ASW1) - $E2D701
- Assign Pointer to Section Part (ASW2) - $E2D702
- Assign Pointer to External Part (ASW3) - $E2D703
- Assign Pointer to Debug Part (ASW4) - $E2D704
- Assign Pointer to Data Part (ASW5) - $E2D705
- Assign Pointer to Trailer Part (ASW6) - $E2D706
- Assign Pointer to Module End (ASW7) - $E2D707

AD Extension Part (ASW0)

- Variable Attributes (NN) - $F0
- Variable Attributes (ATN) - $F1CE
- Variable Values (ASN) - $E2CE †

Environment Part (ASW1)

- Variable Attributes (NN) - $F0
- Variable Attributes (ATN) - $F1CE
- Variable Values (ASN) - $E2CE †

Section Definition Part (ASW2)

- Section Type (ST) - $E6
- Section Alignment (SA) - $E7
- Section Size (ASS) - $E2D3
- Section Base Address (ASL) - $E2CC
- Variable Values (ASR) - $E2D2 †
- Define Context (NC) - $FB
- Physical Region Size (ASA) - $E2C1
- Physical Region Base Address (ASB) - $E2C2
- Mau Size (ASP) - $E2C6
- M-Value (ASM) - $E2CD

† This record type is permissible in this part but it is not yet implemented.

External Part (ASW3)

Public (External) Symbol (NI) - $E8
Variable Attribute (ATI) - $F1C9
Variable Values (ASI) - $E2C9
Variable Values (ASR) - $E2D2 †
External Reference Name (NX) - $E9
External Reference Relocation Information (ATX) - $F1D8
Weak External Reference (WX) - $F4

Debug Information Definition Part (ASW4)

Declare Block Beginning (BB) - $F8
Declare Type Name, file name, line numbers, function name, variable names, etc. (NN) - $F0
Define Type Characteristics (TY) - $F2
Variable Attributes (ATN) - $F1CE
Variable Values (ASN) - $E2CE
Variable Values (ASR) - $E2D2 †
Declare Block End (BE) - $F9

Data Part (ASW5)

Current Section (SB) - $E5
Current Section PC (ASP) - $E2D0
Load Constant MAUs (LD) - $ED
Initialize Relocation Base (IR) - $E3
Repeat Data (RE) - $F7
Variable Values (ASR) - $E2D2 †
Variable Values (ASW) - $E2D7
Load With Relocation (LR) - $E4
Load With Translation (LT) - $FA

Trailer Part (ASW6)

Execution Starting Address (ASG) - $E2C7

Module End (ASW7)

Module End (ME) - $E1
Checksum Records - $EE, $EF

For a description of the Library Information Area, see Appendix D.

† This record type is permissible in this part but it is not yet implemented.

3.1 Header Part

The header part contains information pointing to the location of other parts within the file.

3.1.1 Module Begin (MB)

The MB record must be the first record in the module.

format: {$E0}{Id1}{Id2}

where:

$E0 Record type
Id1 Processor (e.g. ‘‘68000’’ or ‘‘LIBRARY’’)
Id2 Module name

Processor names are listed in Table 3-1.

Name (Id1)	Processor Family	Name (Id1)	Processor Family
29000	AMD 29000	**80C652**	Intel 80C652
1750A	Fairchild 9450, MDC-281, Pace 1750A	**M7700**	Mitubishi MELPS 7700
H8/300	Hitachi H8/300	**68008**	Motorola 68008
H8/500	Hitachi H8/500	**68010**	Motorola 68010
8044	Intel 8044	**68012**	Motorola 68012
8051	Intel 8051	**68020**	Motorola 68020
8052	Intel 8052	**68030**	Motorola 68030
8085	Intel 8085	**68040**	Motorola 68040
80960CA	Intel 80960CA	**68HC32**	Motorola 68HC32
80960KA	Intel 80960KA	**T900**	Toshiba TLCS-900 (controller)
80960KB	Intel 80960KB	**T9000**	Toshiba TLCS-9000 (controller)
80960MC	Intel 80960MC	**TX1**	Toshiba TX1 processor
80C451	Intel 80C451	**TX2**	Toshiba TX2 processor
80C552	Intel 80C552	**64180**	Zilog Z64180 and Hitachi HD64180
80C562	Intel 80C562	**Z80**	Zilog Z80
68000	Motorola 68000	**LIBRARY**	Library Object

Table 3-1. Processor Names

3.1.2 Address Descriptor (AD)

The AD record describes the characteristics of the target processor.

format: {$EC}{n1}{n2}[a]

where:

$EC	Record type
n1	Number of bits/MAU
n2	Number of MAUs constituting the largest address form
a	Optional definition for low order byte significance

$CC	('L') - Low address of field contains least significant byte
$CD	('M') - Low address of field contains most significant byte (default)

Example: The 68000 record will be encoded as $EC0804CD or, equivalently, $EC0804.

3.1.3 Assign Value To Variable W0 (ASW0)

The ASW0 record contains a file byte offset pointer to the AD Extension record relative to the beginning of the file. A zero (0) value indicates that this extension is not included in the file.

format: {$E2}{$D7}{00}{n}

where:

n	Byte offset in file in number format (see Section 2.2)

3.1.4 Assign Value To Variable W1 (ASW1)

The ASW1 record contains a file byte offset pointer to the Environmental record relative to the beginning of the file. A zero (0) value indicates that this extension is not included in the file.

format: {$E2}{$D7}{01}{n}

where:

n	Byte offset in file in number format (see Section 2.2)

3.1.5 Assign Value To Variable W2 (ASW2)

The ASW2 record contains a byte offset pointer to the module Section part relative to the beginning of the module. A zero (0) value indicates that this part is not included in the module.

format: {$E2}{$D7}{$02}{n}

where:

n	Byte offset in file in number format (see Section 2.2)

3.1.6 Assign Value To Variable W3 (ASW3)

The ASW3 record contains a byte offset pointer to the module External part relative to the beginning of the module. A zero (0) value indicates that this part is not included in the module.

format: {$E2}{$D7}{$03}{n}

where:

n Byte offset in file in number format (see Section 2.2)

3.1.7 Assign Value To Variable W4 (ASW4)

The ASW4 record contains a byte offset pointer to the module Debug Information definition part relative to the beginning of the module. A zero (0) value indicates that this part is not included in the module.

format: {$E2}{$D7}{$04}{n}

where:

n Byte offset in file in number format (see Section 2.2)

3.1.8 Assign Value To Variable W5 (ASW5)

The ASW5 record contains a byte offset pointer to the module Data part relative to the beginning of the module. A zero (0) value indicates that this part is not included in the module.

format: {$E2}{$D7}{$05}{n}

where:

n Byte offset in file in number format (see Section 2.2)

3.1.9 Assign Value To Variable W6 (ASW6)

The ASW6 record contains a byte offset pointer to the module Trailer part relative to the beginning of the module. A zero (0) value indicates that this part is not included in the module.

format: {$E2}{$D7}{$06}{n}

where:

n Byte offset in file in number format (see Section 2.2)

3.1.10 Assign Value To Variable W7 (ASW7)

The ASW7 record contains a byte offset pointer to the ME record relative to the beginning of the module.

format: {$E2}{$D7}{$07}{n}

where:

n Byte offset in file in number format (see Section 2.2)

3.2 AD Extension Part

The AD Extension Part contains information describing how the object module was created. This part is located after the header part and the AD record. It is optionally included and is pointed to by the W0 portion of ASW0 if it exists. An NN record with a unique index associates ATN records defining the additional information. For more information on the syntax of records in the AD Extension Part, see Appendix C. The AD Extension Part has the following format:

format:
NN: {$F0}{n1}{Id}
ATN: {$F1}{$CE}{n1}{n2}{n3}[x1][x2][Id]

where:

$F0	NN record type
n1	Symbol name (NN record) type
Id	Symbol name
$F1CE	ATN Record type
n1	Symbol name index (must be the same index as was specified for the NN record)
n2	Symbol type index (unused, set to 0)
n3	Attribute definition: The attribute definitions for the AD Extension Part appear in Table 3-2.

n3	Description
37	Object format version number; requires two extra *numeric* fields [x1] and [x2] defining the version number and revision level respectively. There must not be an ASN record.
38	Object format type; requires one extra field [x1] defining the type : 1 Absolute (not relinkable) 2 Relocatable 3 Loadable 4 Library There must not be an ASN record.
39	Case sensitivity; requires one extra field [x1]. 1 Treat all symbols as if they were upper case 2 Do not change the case of symbols There must not be an ASN record.

Table 3-2. (part 1 of 2) Attribute Definitions for the AD Extension Part

n3	Description
40	Memory model; requires one extra field [x1] defining the memory model. 0 tiny. Code and data are in the same single 64K segment/page. 1 small. Code and data each have a single 64K segment/page. 2 medium. Data has a single 64K segment/page, while code has multiple 64K segments/pages. 3 compact. Data has multiple 64K segments/pages, while code has a single 64K segment/page. 4 large. Both data and code have multiple 64K segments/pages. 5 big. Code has multiple 64K segments/pages, while there is a common "near" data area with far data areas available; normaly data and stack are together 6 huge. All large arrays and structures are in their own section so that addressing involves computations (you can have arrays and structures bigger than 64K) There must not be an ASN record.

Table 3-2. (part 2 of 2) Attribute Definitions for the AD Extension Part

3.3 Environmental Part

The Environmental Part contains information relating to the host environment where the object module was created. It is located after the Header Part and is pointed to by the W1 portion of ASW1. The organization of this part is similar to the AD Extension part described earlier. For more information on the syntax of records in the Environmental Part, see Appendix C. The ATN records have the following format:

format:
NN: {$F0}{n}{Id}
ATN: {$F1}{$CE}{n1}{n2}{n3}[x1[x2[x3[x4[x5[x6[Id]]]]]]]

where:

$F0	NN record type
n1	Symbol name (NN record) type
Id	Symbol name
$F1CE	ATN record type
n1	Symbol name index (must be same index as specified for its associated NN record)
n2	Symbol type index (0 = unspecified)
n3	Attribute definition: The attribute definitions for the Environmental Part appear in Table 3-3 below.

n3	Description
50	Creation date and time; requires one extra field [x1[x2[x3[x4[x5[x6]]]]]]: x1 Year (e.g., 1989) x2 Month (1 - 12) x3 Day (1 - 31) x4 Hour (0 - 23) x5 Minute (0 - 59) x6 Second (0 - 59) The year is encoded as a decimal number, not four hex digits. There must not be an ASN record.
51	Command line text; requires one extra field [Id] containing the command line. There must not be an ASN record.
52	Execution status; requires one extra field [x1]: 0 Success 1 Warning(s) 2 Error(s) 3 Fatal error(s) There must not be an ASN record.

Table 3-3. (part 1 of 2) Attribute Definitions for the Environmental Part

n3	Description
53	Host environment; requires one extra field [x1]: 0 Unknown 1 VMS 2 MS-DOS 3 UNIX 4 HP-UX There must not be an ASN record.
54	Tool and version number used to create the module; requires three extra numeric fields [x1], [x2], and [x3] defining the tool, version, and revision number. Supported tool codes for the [x1] field are listed in each processor-specific appendix. An optional fourth field [x4] defines the tool revision level (e.g., A, B, etc.). Field [x4] is encoded as a single IEEE-695 letter in the range $C1-$DA. There must not be an ASN record.
55	Comments; requires one extra field [Id] specifying the comment string. There must not be an ASN record.

Table 3-3. (part 2 of 2) Attribute Definitions for the Environmental Part

3.4 External Part

The External part contains records used to define and to resolve references for symbols in different modules when they are combined by linking. Variable miscellaneous records are also allowed in the External part. For more information on the syntax of records in the External and Public parts, see Appendix C.

3.4.1 Public (External) Symbol (NI)

The Public Symbol provides for Public symbol definition and is optionally included in a module. Public symbol indices begin at 32. Indices 0 through 31 are reserved.

format: {$E8}{n}{Id}

where:

$E8	Record type
n	Public name index number, unique within an object file (must be > 31, 0 - 31 reserved)
Id	Symbol name

3.4.2 Attribute Records (ATI)

format: {$F1}{$C9}{n1}{n2}{n3}[x1][x2][x3][x4]{n4}

where:

$F1C9 ATI record type
n1 Symbol name index (this must be the same index as specified for the NI record)
n2 Symbol type index as follows:

0	Unspecified
3	8-bit data byte
5	16-bit short data word
7	32-bit long data word
10	32-bit floating point
11	64-bit floating point
12	10 or 12 byte floating point
15	Instruction address

n3 Attribute definition: The attribute definitions are described in Table 3-4.
n4 If n2 is non-zero, number of elements in the symbol type specified in n2

n3	Description
8	Global compiler symbol. There must be an ASI record specifying the address/value.
16	Constant, with the following additional fields. x1 Symbol class, required. Defined as follows. 0 Unknown class. 1 EQU constant. 2 SET constant. 3 Pascal CONST constant. 4 C #define constant. 5-128 Reserved for future use. x2 Public/local indicator, optional. Omitted or zero means local. One means public. x3 Numeric value, optional. For constants with ordinal values. Either x3 or x4 should be present but not both. x4 String value, optional. For constants with string values. For each such ATI record, there may be one ASI record indicating the program counter address where the definition occurred. If such an ASI record is not present, then the constant value is assumed to be valid everywhere in the object module. There may be more than one ATI/ASI record pair for a single name since constants defined via SETs and #defines may be redefined repeatedly.
19	Static symbol generated by assembler. There must be an ASI record specifying the address/value.

Table 3-4. Attribute Definitions for the External Part

3.4.3 Value Records (ASI)

The ASI record defines values for variables.

format: {$E2}{$C9}{n1}{n2}

where:

$E2C9 Record type
n1 Symbol index (this must be the same index as specified for the NI record)
n2 Expression defining value for symbol

The expression typically involves a section base for addresses and not for constants.

Example: {$E2}{$C9}{$02}{$D2}{$05}{$10}{$A5} is the binary representation of the character form ASI2,R5,10,+. This assigns the value ‘‘variable offset of the section’’ whose section index is 5, plus offset $10 to variable I2. I2 is the public symbol whose public name index is 2.

3.4.4 External Reference (NX)

The NX record references the name of a symbol in another module. External Reference indices begin at 11. Indices 0 - 10 are reserved for special case symbols such as register designators.

format: {$E9}{n1}{Id}

where:

$E9	Record type
n1	External reference index unique to this module
Id	Symbol name

3.4.5 External Reference Information (ATX)

The ATX record contains additional definition information for an External Reference symbol. This record is optionally generated.

format: {$F1}{$D8}{n1}[n2][n3][n4]

where:

$F1D8	Record type
n1	External reference index (this must be the same index as specified for the NX record)
n2	Type index (0 = unspecified)
n3	Section index
n4	Short external flag (0 = not short)

Optional numbers can be omitted but if a later one is present, the omitted number must be filled with the ‘‘omitted number’’ construct {$80}. The defaults for omitted numbers are: no type checking, no section checking, and the external is not necessarily in ‘‘short’’ form.

3.4.6 Weak External Reference (WX)

A weak external symbol is a global symbol which may be declared in more than one of the constituent modules of a composite (linked) object module. Linkers treat weak externals like external references, except during a link to produce an absolute object module. In the latter case, if no explicit external definition exists for the weak external symbol in any of the constituent modules, the linker will allocate in the Section Part the largest space required for the weak external symbol among all of its instances, and will create a public definition for the weak external symbol in the Public/External Part.

The basic properties of a weak external symbol, e.g., name, type, etc., are described in an NX/ATX record pair in a manner similar to that used for normal external references. In addition, a WX record having the same index as the accompanying NX/ATX pair further identifies the symbol as a weak external symbol.

format:{$F4}{n1}{n2}[n3]

where:

$F4	Record type for WX
n1	External reference index (must be same as in NX record)
n2	Default size if not resolved (0 = unspecified; defaults to int for the target processor)
n3	Default value if not resolved (0 = unspecified).

Some consuming tools require a mechanism to identify the module that declared each public symbol. In object modules containing several constituent modules, ownership of normal public symbols can be determined by finding which module claims ownership of the address range containing the space allocated to the symbol. (See BB11 blocks).

In relocatable modules produced by translation of a single source file, there is no question of the ownership of a weak external, because the resulting relocatable object module has only one constituent module.

Expressing the ownership of weak externals in object modules having several constituent modules presents a more difficult problem, however. For example, during incremental linking, i.e, producing a relocatable object module by combining multiple relocatable object modules, ownership of a weak external is lost because there are no provisions for expressing scoping in the Public/External Part. Also, when the linker allocates the space for a weak external in the process of creating an absolute object module, it is certain that the allocated space will not be associated with any of the constituent modules, obviating the use of the symbol's address as a proxy for the owning module.

To convey to consuming tools the identity of the module that declared a weak external symbol, the linker generates NN and variable miscellaneous ATN records having code 62 in the Public/External Part. The name of the module owning the weak external is specified in the NN record. There may be multiple variable miscellaneous ATN code 62 records, each representing a specific weak external declaration in some module, pointing to the single NN record for that module. The detailed syntax of variable miscellaneous ATN record type 62 is explained in Appendix B.

3.5 Section Part

The Section part contains information defining the sections of the module. A ‘‘section’’ in this context is a contiguous area of memory. It may be absolute or relocatable, and may or may not have a name. A section is absolute if and only if the ‘‘AS’’ attribute is specified in its ST record (see below). All data MAUs must be defined in a section.

Relocatable modules are produced by the assembler or the compiler. A relocatable module can have named relocatable sections and both named and unnamed absolute sections. An absolute module must have only named or unnamed absolute sections.

Symbol definitions for addresses are relative to a section and definitions for constants are absolute for the relocatable format. In absolute format, all symbol definitions are absolute.

For more information on the syntax of records in the Section Part, see Appendix C.

3.5.1 Section Type (ST)

Each section must have exactly one section type record.

Section types may appear in any order, except that no forward references are allowed: sections must be defined before they are referred to in ‘‘parent’’ or ‘‘brother’’ fields of other sections.

SA, ASA, ASB, ASF, ASL, ASM, ASR, and ASS records must appear after the ST record they refer to.

format: ${E6}{n1}{l}[Id][n2][n3][n4]

where:

$E6 Record Type
n1 Section index (index must be greater than zero and unique to this module)
l Section type (only the new section types are described here)

AS {$C1}{$D3} normal attributes for absolute sections. Sections from different modules with these attributes, whether they have the same name or not, are considered to be unrelated.

ASP {$C1}{$D3}{$D0} absolute code
ASR {$C1}{$D3}{$D2} absolute ROM data
ASD {$C1}{$D3}{$C4} absolute data

B A section which can contain other sections.

- A B section has a logical address and size which must contain the address ranges of all of its children.
- A B section may specify the upper half of the logical address (M-value or memory space number) for itself and its children.
- A B section may specify the physical addresses for itself and its children using a sequence of ASA and ASB records all referring to the same B section. This sequence defines one or more ‘‘hunks’’ of physical memory.
- A B section may specify a context. There may be more than one level of B section.

- A B section may be define a MAU (Minimum Addressable Unit) for itself and its children which has a different size than the standard MAU for the target processor.
- A B section is not loadable. That is, it does not contain data which is put into memory.

The addresses of the B section map onto the addresses of the physical hunks as described below. Let L refer to the logical lowest address of this section. Let A(i) refer to the length specified by the ith ASA record for this section. Let B(i) refer to the address specified by the ith ASB record. Let T(i) be the sum of the sizes of all the previous i-1 hunks.

```
T(i) = A(1) + A(2) + ... + A(i-1)
T(1) = 0
```

Then the logical range of addresses

```
L+T(i) through L+T(i)+A(i)-1
```

map onto the physical addresses

```
B(i)   through B(i)+A(i)-1
```

The ASB record must immediately follow the ASA record for a particular hunk. If the ASB record is omitted (indicated by two ASA records in a row) the hunk is unmapped (i.e. represented by no physical memory).

The total of the ASA records for a section must be less than or equal to the value of the ASS record for the section. If there are fewer physical MAUs than logical MAUs, then the leftover logical MAUs are unmapped (i.e. represented by no physical memory).

C {$C3} normal attribute for named relocatable sections. Sections from different modules with the same name will be concatenated at link time. This section may also be explicitly typed as:

CP {$C3}{$D0} normal code
CR {$C3}{$D2} normal ROM data
CD {$C3}{$C4} normal data

E {$C5} or M {$CD} shared (common) data sections. M specifies that different-sized sections from different modules be merged into one section whose size is the maximum of the input sizes. E specifies that an error condition exists if common sections are not the same size.

EA {$C5}{$C1} Common absolute sections. EA sections from different modules are overlayed at the same absolute address. It is an error if AE sections from different modules have different sizes. It is an error if the AE sections from different modules have different addresses. This section may be explicitly typed as:

EAP {$C5}{$C1}{$D0} common absolute code
EAR {$C5}{$C1}{$D2} common absolute ROM data
EAD {$C5}{$C1}{$C4} common absolute data

EZ {$C5}{$DA} the attribute for short common with error checking.

T A section which provides an alternate view of the memory described by its brother section. T sections allow different logical addresses to access the same physical memory within a context. The brother section must have type B. T sections allow the same physical memory to appear in more than one context with either equal or differing logical addresses. T sections may not have ASA or ASB records.

A T section has all the properties of its brother (n3) except perhaps for the following: the parent (n2), the logical address (specified by ASL and ASM records), its children, or its MAU size (specified by the ASF record). In particular, it has the same physical size and the same physical mapping as its brother. (Note: the physical size is figured using both the S-value and MAU size factor (F-value) for a section. See the 8051 example.)

Two or more T sections may refer to the same brother.

ZC {$DA}{$C3} the attribute for ‘‘short’’ relocatable sections other than common. This section may also be explicitly typed as:

ZCP {$DA}{$C3}{$D0} short code
ZCR {$DA}{$C3}{$D2} short ROM data
ZCD {$DA}{$C3}{$C4} short data

ZM {$DA}{$CD} the attribute for ‘‘short common’’ relocatable sections.

Id Section name. (Not necessarily unique within a context.)

n2 Section index of the parent. Zero or omitted means none. Any type of section may have a parent. No section may be an ancestor of itself.

n3 Section index of a brother. Zero or omitted means none. Only type T sections may have a brother. The brother must have type B.

n4 Context index. Zero or omitted means none. Only a B section or a T section may specify a context.

Examples:
{$E6}{$02}{$C3}{$04}‘‘CODE’’
{$E6}{$03}{$C5}{$06}‘‘COMMON’’

A consuming tool (e.g., a debugger) may need to determine the content of some region of the memory map so that the tool’s behavior can adapt in a manner appropriate to the content of the section. For example, the debugging tool may be designed to refuse to begin execution at a memory address in a section containing data.

The content of a section is specified in two places: in the section content modifiers $D0 (code), $D2 (ROM data) and $C4 (data) attached to the section type field (l) of the ST record, and in the section type {n2} field of the BB11 blocks in the Debug Part. Consuming tools need to interpret the section content information with care.

The section content modifiers of the ST records of absolute (linked) object modules may incorectly indicate the contents of a section and may thus be misleading to a consuming tool when, for example, when the linker is instructed to combine sections having differing content (e.g., code and data) into a single parent section. In this case, the content of the parent section as portrayed in the ST record is indeterminate.

However, the original content of the parent section’s constituent subsections, is preserved in the section type {n2} field of the individual BB11 blocks of the Debug Part. There is one BB11 block for each of the children of the parent section, and the BB11 blocks are required to be present even in object modules stripped of symbolic debugging information. (See Section 3.6.1 "Block Begin (BB)".)

3.5.2 Define Context (NC)

The NC record defines a context. An NC record must be inside the Section Part.

Define Context records may appear in any order. In particular, they may appear after the ST records which refer to them.

format: ${FB}{n1}[Id]

where:

$FB	Record Type
n1	Context Index (index must be greater than 0 and unique to this module)
Id	Context Name (The context name need not be unique withing a module)

There are no variables associated with a context.

3.5.3 Section Alignment (SA)

The SA record defines the boundary alignment and boundary crossing parameters for relocatable sections.

format: {$E7}{n1}[n2][n3]

where:

- $E7 Record type
- n1 Section index (this must be the same index as specified for the ST record)
- n2 Boundary alignment divisor
  - 0 Processor default (e.g., 4 bytes for the 68000, 1 byte for the Z80)
  - 1 Byte
  - 2 Word
  - 4 Long (quad)
  - n Any power of 2
- n3 Page size - if present, align the section to the next multiple of n3 if it does not fit below the next multiple of n3. n3 must be a power of 2. Value must be in MAUs.

3.5.4 Section Size (ASS)

The ASS record is required for all sections and defines the size for this section.

format: {$E2}{$D3}{n1}{n2}

where:

- $E2D3 Record type
- n1 Section index. An ST record must have occurred before this ASS record.
- n2 Section size (in MAUs). This expression must be a simple number.

3.5.5 Physical Region Size (ASA)

An ASA record defines the size of a region of physical memory. The region corresponds to or maps all or part of a B section.

More than one ASA record may exist for a section. This indicates that the section maps onto more than one physical region. The order of ASA records is significant. The first addresses of the section map onto the bytes map onto the second region until those bytes are all used; and so forth.

If several ASA for a single section exist, no other records may occur between the ASA records except for (possibly) one ASB record after an ASA record.

The total of all A-values for a section must be less than or equal to the logical size (S-value) for the section. If there are fewer physical MAUs than logical MAUs, then the leftover logical MAUs map to no physical memory.

format: {$E2}{$C1}{n1}{n2}

where:

- $E2C1 Record type
- n1 Section index. An ST record with the same index must have occurred before this ASA record. The ST record must have type B.
- n2 Region size in MAUs. This expression must be a simple number.

3.5.6 Physical Region Base Address (ASB)

An ASB record defines the beginning address of a region of physical memory. The region corresponds to or maps all or part a B section. More than one ASB record may exist for a section. The ASB record must occur immediately following the ASA record which defines the size of the region. If an ASB record does not occur directly after an ASA record, it indicates that the physical region does not exist. That is, the corresponding addresses of logical memory map to no physical memory.

format: {$E2}{$C2}{n1}{n2}

where:

$E2C2	Record type
n1	Section index. An ST record with the same index must have occurred before this ASB record. The ST record must have type B.
n2	Physical address of the region.

3.5.7 MAU Size (ASF)

The ASF record defines a new MAU (Minimum Addressable Unit) in terms of the target processor MAU.

format: {$E2}{$C6}{n1}{n2}

where:

$E2C6	Record type
n1	Section index. An ST record with the same index must have occurred before this ASF record. The ST record must have type B or T.
n2	MAU size. If this number is positive, it indicates the section MAU is smaller than the target MAU. n2 section MAUs make one target MAU. If this number is negative, it indicates the section MAU is larger than the target MAU. -n2 target MAUs make one section MAU. This expression must be a simple number.

3.5.8 M-value (ASM)

The M-value is the most significant half of the two part logical address. (The least significant half is the L-value). The meaning of the M-value is defined for each target processor that needs a two-part address.

Whether or not an M-value is required as part of the address is defined for each target processor. For example, an MC68000 section with no ASM record for itself or its parent has an ‘‘unspecified’’ M-value.

format: {$E2}{$CD}{n1}{n2}

where:

$E2CD	Record type
n1	Section index. An ST record with the same index must have occurred before this ASM record. The ST record must have type B or T.
n2	M-value

3.5.9 Section Base Address (ASL)

ASL records specify the section base address. Each section defined by an ST record in either relocatable or absolute object modules must have an ASL record, with one exception. The one exception is that relocatable sections in relocatable object modules can have no ASL records.

format: {$E2}{$CC}{n1}{n2}

where:

$E2CC	Record type
n1	Section index (this must be the same index as specified for the ATN record)
n2	Section Base address (in MAUs)

3.5.10 Section Offset (ASR)

An ASR record is generated each time the section offset variable changes.

format: {$E2}{$D2}{n1}{n2}

where:

$E2D2	Record type
n1	Section index
n2	Expression defining a new section offset (in MAUs)

3.6 Debug Information Part

The Debug Information part contains records that define how to determine the symbol related information for a module at execution time. This is required for debuggers that provide high-level debugging capabilities.

For information on the syntax of records in the Debug Information Part, see Appendix C.

3.6.1 Block Begin (BB)

The BB records are an extension to the standard. They provide definitions of debugging information related to the high level language definitions for typedef, scoping, and line numbers. They also provide assembly level language definitions for modules and local symbols. A block beginning with a BB is terminated with a BE record. BB records can be nested according to rules described below. Nested BB blocks can be used to capture scoping information. The types of BB blocks include:

BB1	Type definitions local to a module.
BB2	Type definitions global to all modules.
BB3	A module. A non-separable unit of code, usually the result of a single compilation, i.e. a Modula-2 module or an Ada package.
BB4	A global subprogram.
BB5	A source file line number block.
BB6	A local (static) subprogram.
BB10	An assembler debugging information block.
BB11	The module portion of a section.
BB20	Library - contains a list of global symbols used in a module. For a description of the Library Information Area, see Appendix D.

The following list describes features of some of the blocks.

- There can be at most one BB2 block in an object file and it must occur before any other BB records.
- BB2 blocks are intended to collect all global type information in a single place. Support for BB2 blocks is not currently implemented in most HP/MRI assemblers and linkers. Thus redundant type definitions, (e.g., from a common #include file) are output in each module type block (BB1) where they occur. This is not an optimal use of object file bytes, but is perfectly acceptable as far as object file readers are concerned.
- BB1, BB3 and BB5 blocks usually occur together and in that order.
- BB1 blocks can be absent for modules which declare no local types.
- Although it is not required, BB5 blocks immediately follow BB3 blocks in this implementation.
- A BB5 cannot occur without a BB3.
- Consecutive BB3 and BB5 blocks must refer to the same module.
- BB3s and BB6s can be arbitrarily nested in one another.

Block Nesting

Nested BB3s can occur only in Ada modules. BB1s and BB5s cannot be nested in BB3s, BB4s, or BB6s, however, BB5s can be nested in BB5s. For a summary of block nesting rules, see Table 3-5 below.

```
Global Type Definitions (BB2)
        NN and TY records
Global Type Definitions End (BE2)
Module-Scope Type Definitions (BB1)
        NN and TY records
Module-Scope Type Definitions End (BE1)

High Level Module Block Begin (BB3)
        Global Variables (NN, ATN8, ASN)
        External Variables (NN, ATN5, ASN)
        External Functions (NN, ATN4, ASN)
        Module-Scope Variables (NN, ATN3, ASN)

        Module-Scope Function Block Begin (BB6)
                Local Variables    (NN, ATN, ASN)
        Module-Scope Function Block End (BE6)

        Global Function Block Begin (BB4)
                Local Variables    (NN, ATN, ASN)

                Local Function Block Begin (BB6)
                        Local Variables    (NN, ATN, ASN)
                Local Function Block End (BE6)

        Global Function Block End (BE4)                    } High Level Module Block
                                                             (one for each High Level Module)
High Level Module Block End (BE3)

Source File Block Begin     (BB5)
        NN, ATN, ASN, for line numbers in main source
        BB5 - Included File
                NN, ASN, ATN for line numbers
        BE5
        NN, ASN, ATN, for line numbers in main source
Source File Block End (BE5)

Assembly Module Block Begin (BB10)
        Compiler Generated Global/External
                Variables (NN, ATN19, ASN)                 |
        Compiler Generated Local Variables (NN, ATN19, ASN)|

        Assembler Section Block Begin (BB11)
        Assembler Section Block End (BE11)

        Assembler Section Block Begin (BB11)
        Assembler Section Block End (BE11)

Assembly Module Block End (BE10)
```

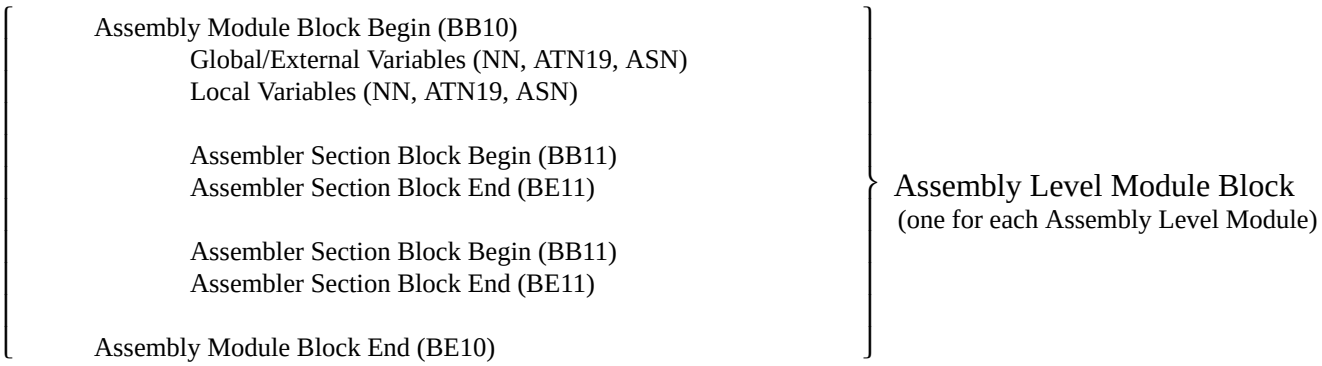

Table 3-5 below illustrates which of the blocks under **Inner** can be nested within the blocks listed under **Outer**. Some of the blocks require an outer block. For example, a BB4 block requires that its outer, enclosing block be a BB3. Similarly, a BB1 or BB2 block requires that its outer, enclosing block be the Debug Part, or **debug**.

Inner	**Outer**								
	BB1	**BB2**	**BB3**	**BB4**	**BB5**	**BB6**	**BB10**	**BB11**	**debug**
BB1	no	no	no	no	no	no	no	no	yes
BB2	no	no	no	no	no	no	no	no	yes
BB3	no	no	yes (Ada)†	yes (Ada)	no	yes (Ada)	no	no	yes
BB4	no	no	required	no	no	no	no	no	no
BB5	no	no	no	no	yes	no	no	no	yes
BB6	no	no	yes	yes	no	yes	no	no	no
BB10	no	no	no	no	no	no	yes	no	yes
BB11	no	no	no	no	no	no	required	no	no

Table 3-5. Summary of Permitted Block Nesting

Block Size

There are different formats for BB records which require information depending on block type. In all cases, the block size in bytes refers to the number of bytes from the beginning of the BB record to the end of the corresponding BE record. This size can be used by a reader to skip ahead in the file, rather than having to read through it. For example, in the segment below, the BB record is specified as being $30 bytes long:

```
0000 BB 1 $30 test_mod
.
.
.
002f BE
0030 BB 3 . . .
```

The size, $30, will be added to the address of the beginning of the BB 1 record to obtain the seek address of the BB3.

---

† The combinations marked (Ada) are permissible for Ada but not yet implemented.

The format for each block type is described below:

Block Type 1 - unique typedefs for module

format: {$F8}{$01}{n1}{Id}

where:

$F8	Record type
$01	Block Type 1 - unique typedefs for module
n1	Block size in bytes (0 = unknown)
Id	Module name

Block Type 2 - global typedefs

format: {$F8}{$02}{n1}{Id}

where:

$F8	Record type
$02	Block Type 2 - global typedefs
n1	Block size in bytes (0 = unknown)
Id	Zero length name

Block Type 3 - high level module scope beginning

format: {$F8}{$03}{n1}{Id}

where:

$F8	Record type
$03	Block Type 3 - high level module scope beginning
n1	Block size in bytes (0 = unknown)
Id	Module name (must be the same name as specified for BB1)

Block Type 4 - global function

format: {$F8}{$04}{n1}{Id}{n2}{n3}{n4}

where:

$F8	Record type
$04	Block Type 4 - global function
n1	Block size in bytes (0 = unknown)
Id	Function name
n2	Number of bytes of stack space required for local variables (in MAUs)
n3	Type index for return value parameter and function information ('x' type), (0 = unknown)
n4	Offset expression (in MAUs). In relocatable object modules, the offset is expressed in terms of R-variables. In absolute object files, the offset expressions are the absolute addresses of the beginning of the code block.

Block Type 5 - file name for source line numbers

format: {$F8}{$05}{n1}{Id}[n2[n3[n4[n5[n6[n7]]]]]]

where:

$F8	Record type
$05	Block Type 5 - file name for source line numbers
n1	Block size in bytes (0 = unknown)
Id	Source file name (including path)
n2	Year (e.g., 1988)
n3	Month (1-12)
n4	Day (1-31)
n5	Hour (0-23)
n6	Minute (0-59)
n7	Second (0-59)

Block Type 6 - local function

format: {$F8}{$06}{n1}{Id}{n2}{n3}{n4}[Id]

where:

$F8	Record type
$06	Block Type 6 - local function (static)
n1	Block size in bytes (0 = unknown)
Id	Function name
n2	Number of bytes of stack space required for local variables (in MAUs)
n3	Type index for return value parameter and function information ('x' type) (0 = unspecified)
n4	Offset expression (in MAUs). In relocatable object modules, the offset is expressed in terms of R-variables. In absolute object files, the offset expressions are the absolute addresses of the beginning of the code block.

If the function name does not exist (length = 0), this is an unnamed block used for variable scoping only.

Block Type 10 - assembler module scope beginning

format: {$F8}{$0A}{n1}{Id}{Id}{n2}[Id][n3[n4[n5[n6[n7[n8]]]]]]

where:

$F8	Record type
$0A	Block Type 10 - assembler module scope beginning
n1	Block size in bytes (0 = unknown)
Id	Module name or file name (with path)
Id	Input relocatable object file name (used during incremental linking only, zero length string otherwise)
n2	Tool type: To find the code for the tool type, see the appendix on Family Definitions for your target.
Id	Version and revision in string format.
n3	Year (e.g., 1988)
n4	Month (1-12)
n5	Day (1-31)
n6	Hour (0-23)
n7	Minute (0-59)
n8	Second (0-59)

If the BB 10 block did not exist in the relocatable module, the linker will create a dummy BB10 based on the module name to cover backwards compatibility.

The first Id field in the BB10 record holds one of the following:

A module name. Normally, the module name is the assembler source file name with the suffix and directory path stripped. An assembler directive can override this default. In the case of BB10 blocks associated with high level modules (e.g., BB1, BB3, BB5, or BB10) the module name in the BB10 field must match the module name in the associated BB3 and BB1 blocks. Also, BB10 blocks for modules assembled without debugging turned on hold only the module name.

The path name of the assembly source file. This applies to BB10 blocks for hand coded assembly source files.

Block Type 11 - module section

format: {$F8}{$0B}{n1}{Id}{n2}{n3}{n4}{n5}

where:

$F8	Record Type
$0B	Block Type 11 - module section
n1	Block size in bytes (0 = unknown)
Id	Zero length name (section name already defined)
n2	Section type

0	Mixture of code, data, etc.
1	Code
2	Read/Write data
3	Read only data
4	Stack
5	Memory

n3	Section index (adjusted by linker when combining sections)
n4	Offset expression (in MAUs) - the expression is in terms of R-variables for relocatable files. For absolute files, it is an absolute address.
n5	

0	Map section to HP ABS
1	Map section to HP PROG
2	Map section to HP DATA
3	Map section to HP COMM

In the BB11 block, the parser stop (i.e., binary comma) code $90 is required before the HP Section mapping information, parameter {n5}.

Optional fields may be null; but if any field is null and a later field is present, the omitted field must be filled with the {$80} construct. The relationship of blocks to variable attribute and variable value records (NN, ASN, ATN records) is preserved in the file. For variables which have an NN, ASN, ATN triple, these records must be together in the block structure definition (i.e., there can be no BB nor BE records between them). Block definitions may be nested.

3.6.2 Variable Names (NN)

These NN records declare variable names, type names and line numbers. The IEEE standard has been extended to allow duplicate local symbols to be defined, as long as the indices and the scoping are different. This provides symbol definitions that are local to a specific section.

format: {$F0}{n}{Id}

where:

$F0	Record type
n	Name index number (must be > 31, 0-31 are reserved)
Id	Name

3.6.3 Define Types (TY)

The TY record specifies that a type name represents an explicit type definition other than the implicit types predefined for use with MRI/HP language variables. In some languages, such as PASCAL, different types with the same name may be declared. This is supported by this specification by having multiple NN, TY pairs with the same name in the NN.

format: {$F2}{n1}{$CE}{n2}[n3][n4]...

where:

$F2	Record type
n1	Type index unique within module (>255) (0-255 reserved for implicit types)
$CE	Record type
n2	Local name index for symbol defined by NN record
n3,n4...	Variable number of fields specifying additional type information as defined in Appendix A.

3.6.4 Attribute Records (ATN)

Each ATN record (with the exception of ATN 9, see Table 3-6) is associated with an NN record and defines a valid symbol.

format:
NN record: {$F0}{n1}{Id}
ATN record: {$F1}{$CE}{n1}{n2}{n3}[x1][x2][x3][x4][x5][x6][Id]
ASN record: {$E2}{$CE}{n1}{n2}

where:

$F0	NN record type
n1	Symbol name (NN record) type
Id	Symbol name
$F1CE	ATN record type
n1	Symbol name index (this must be the same index as specified for the NN record)
n2	Symbol type index (0=untyped)
n3	The numbers representing the attributes, the blocks they can appear in, and their descriptions are illustrated in Table 3-6 below.
x1 ... Id	Optional features, described for each attribute.
$E2CE	ASN record type
n1	Symbol name (NN record) index
n2	Symbol value

n3	Block	Description
1	4,6	Automatic variable; requires an additional field [x1] defining the stack offset (in MAUs). There must not be an ASN record.
2	4,6	Defines a variable name as a living register; requires one extra field [x1]. This field is the index of the register name. To find the register index, see the appendix on Family Definitions for your target. There must not be an ASN or ASI record.
3	3,4,6	Compiler defined static variable. There must be an ASN or ASI record specifying the address/value. †
4	3,4,6	External function. Definition type is 'x' type. There must not be an ASN record.
5		External variable definition. There must not be an ASN record.
7	5	Line number; requires two extra fields giving the line number and column number. Two optional fields [x3] and [x4] are reserved and should be omitted. The line and column number represent the end of a group of one or more lines in a statement. A column number of 0 represents the end of the line. Otherwise, the column number represents the last character in a statement (e.g., the ";" of a C statement). Line numbers do not have to be in ascending order, and it is the consuming tool's responsibility to handle numbers which are "out of order." There must be an ASN record specifying the address.
8	3	Compiler global variable.†† There must be an ASN record specifying the address/value. See also Section 3.4.2.
9	4,6	Variable life time; this record controls the temporary allocation of symbols to register resources. ATN9 requires one extra field, [x1], which specifies the absolute program counter offset (in MAUs) from the current segment at which the first processor instruction reflecting the change can occur. Parameter {n1} refers either to a previously defined NN/ATN pair ({n1}>0) or to the special reserved NN index 0 ({n1}=0). When {n1} is 0, [x2] is interpreted as the index of the register resource returned to scratch status starting at the program counter offset specified in [x1]. All variable lifetime symbols in this register resource are "dead" starting at the address in [x1]. When {n1}>0, {n1} is interpreted as the index of a local symbol which must have been previously declared in an NN/ATN pair having index {n1}. The original NN/ATN symbol declaration must indicate some form of register storage class. This may be done in any of several ways: 1. Symbol declared as a living register variable. The symbol must have been declared in the immediately enclosing BB4 or BB6 scope. See ATN2. 2. Symbol declared as an automatic variable. The symbol must have been declared in the immediately enclosing BB4 or BB6 scope. See ATN1. 3. Symbol declared as a shadowed variable. The symbol may be any automatic (stack), static, or global variable within the current BB3 (module) block. See variable miscellaneous record 60, Appendix B. 4. Compiler register utilization specified explicitly. Register storage specified for entire classes of variables. See module miscellaneous code 61, Appendix B There must not be an ASN record.
10	4,6	Defines a variable name as a locked register; requires two extra fields, [x1] and [x2], to define the index of the register name and the frame offset (in MAUs). There must not be an ASN record.
11	3,4,6	Reserved for FORTRAN Common.

Table 3-6. (part 1 of 4) Attribute Numbers, Blocks and Descriptions

n3	Block	Description
12	3,4,6	Based variable. It has the following additional fields: x1 Offset value. x2 Control number. 0 Based from static memory. The base is a relocatable expression and evaluates to 0 if omitted. This control value allows basing from another variable (such as external) for aliasing and special languages. If the space argument is used, it can allow reference to other than normal address spaces (such as bit space). 1 Based from register. It is often used for index relative addresses, when the absolute address is not known until run time. 2 Based from bank, section, or task. This allows special addressing for MMU environments. It also handles instanciated data, as is used with ADA. 3 Based from selector or pointer. This allows for indirected addressing. This can be used for conformant arrays and PLM ‘‘based’’ variables. It can also be used for local heap-based statics. 4 Indirected from register base. This allows for indirected addressing. This is commonly used with ‘‘pass by reference’’ languages such as FORTRAN, Pascal ‘‘var’’ args, and ADA ‘‘in out’’ args. Like control 1, it creates an address from register+offset, but the address is treated as a pointer. x3 Public/local indicator. Optional. Omitted or zero means local. One means public. x4 Memory space indicator value - defined for each preprocessor. Default = 0x80. x5 Base_size. The number of MAUs the base value can occupy. The base value, *which is contained in an accompanying ASN record,* is defined by the value of control. 0 Base is an address expression. 1 Base is a register index (predefined per processor). 2 Base is a section or super-section index. 3 Base is an address expression relating to another symbol. If another symbol is not defined at that address, base refers to a default pointer typed object at the specified address. 4 Base is a register index (usually the frame pointer) which when added to the offset computes an address. The resultant address is treated like a normal pointer (for the processor) to the actual object.

Table 3-6. (part 2 of 4) Attribute Numbers, Blocks and Descriptions

† Fortran entry statements are designated by an ATN 3 with an ’x’ type. The ’x’ type defines the argument types. The parameters are defined as locals with the corresponding type and address.

†† May be present as an ATI record in the Public External Part if it is a global symbol.

n3	Block	Description
16	3,4,6,10	Constant, with the following additional fields. x1 Symbol class. Required. Defined as follows: 0 Unknown class. 1 EQU constant. 2 SET constant. 3 Pascal CONST constant. 4 C #define constant. 5-128 Reserved for future use. x2 Public/local indicator. Optional. Omitted or zero means local. One means public. x3 Numeric value. Optional. For constants with ordinal values. Either x3 or x4 should be present but not both. x4 String value. Optional. For constants with string values. There may be more than one ATN for a single name since SETs and #defines may be defined repeatedly. For each such ATN record, there may be one ASN record indicating the program counter address where the definition occurred. If such an ASN record is not present, then the constant value is assumed to be valid between the lowest and highest program counter addresses of the module in which the NN/ATN record pair defining the constant was found.
19	10	Static variable generated by assembler; may be global in scope.† There must be an ASN record specifying the address/value. There is one required field [x1], which indicates the number of elements of type n2 described by the symbol, and [x2], which is a local/global indicator. [x2]=omitted or 0 means local. [x2]=1 means global. See also Section 3.4.2.
36	AD Part	Contains the lowest version number of all input files that were used to build this file (see also AD Extension Part).
37		See Section 3.2
38		See Section 3.2
39		See Section 3.2
50		See Section 3.3
51		See Section 3.3
52		See Section 3.3
53		See Section 3.3
54		See Section 3.3
55		See Section 3.3
62	4,6	Procedure block misc.; followed by two fields which describe the most recent procedure block. The first field [x1] is the pmisc. type identification number, the second [x2] is the number of additional ATN 65 or ASN records associated with this directive. Refer to the appendix on Miscellaneous Records for the codes associated with this directive.

Table 3-6. (part 3 of 4) Attribute Numbers, Blocks and Descriptions

† May be present as an ATI record in the Public External Part if it is a global symbol.

n3	Block	Description
63	3,4,6	Variable misc.; followed by two fields which describe a variable. The first field [x1] is the vmisc. type identification number, the second [x2] is the number of additional ATN 65 or ASN records associated with this directive. Refer to the appendix on Miscellaneous Records for the codes associated with this directive.
64	3	Module misc.; followed by two fields which describe the current module block. The first field [x1] is the mmisc. type identification number, the second [x2] is the number of additional ATN 65 and ASN records associated with this directive. Refer to the appendix on Miscellaneous Records for the codes associated with this directive.
65	3,4,6	Misc. string; requires one field which is a string value for miscellaneous records 62, 63, and 64.

Table 3-6. (part 4 of 4) Attribute Numbers, Blocks, and Descriptions

The appendix entitled *Sample C, and C++ Programs and Their IEEE Format* contains examples of all of the attribute definitions listed above except for 9, 11, 55, 62, 63, and 65.

3.6.5 Value Records (ASN)

The ASN records are used to define values for variables.

format: {$E2}{$CE}{n1}{n2}

where:

$E2CE Record type
n1 Symbol name index (must be the same as specified for the NN record)
n2 Expression defining value for symbol (in MAUs if it is an address)

The expression typically involves a section base for addresses and not for constants.

Example:

{$E2}{$CE}{$02}{$CC}{$05}{$10}{$A5} is the binary representation of the character form ASN2,L5,10,+. This assigns the value ‘‘base of the section’’ whose section index is 5, plus $10 to variable N2, the local symbol whose symbol index is 2.

Stack relative symbols and register-based symbols must not have an ASN record since the value is defined at execution time.

3.6.6 Compiler Id

Compiler Id Codes directly follow the BB3 record. A dummy NN record precedes the initial ATN record in order to produce a symbol name index. One ATN record defines that this is a module miscellaneous directive. It is followed by three ASN records for tool code, type checking code, and default pointer size in MAUs. These are optionally followed by one ATN for the version number and up to six ASNs for the date and time.

format: {$F1}{$CE}{n1}{0}{64}{50}{n5}{ASN1}{ASN2}{ASN3}[ATN1][ASN4[ASN5 [ASN6[ASN7[ASN8[ASN9]]]]]]

where:

$F1CE ATN Record type
n1 Symbol name index produced by an NN record.
0 Symbol type index
64 Attribute definition of 64 for module misc.
50 Module misc. type identification number of 50 (ATN record)
n5 Miscellaneous record count (based on number of date values, etc)
ASN1 Tool code definition. To find the tool code, see the appendix on Family Definitions for your target.
ASN2 Rule mask - a bit mask specifying assorted object module processing rules:

Bit	Meaning
0	Type equivalency rule 0 Opaque type equivalence 1 Transparent type equivalence Note: Opaque type equivalence means that all types defined in the object module are to be treated as unique types even if they are derived from the same type and are the same size. Pascal and Ada have opaque type equivalence. Transparent type equivalence means that if two types are derived from the same underlying type, then they are equal. C has transparent type equivalence.
1	Interpretation of lifetime register variables information. 0 Register lifetime information is provided. 1 Register lifetime information is not provided. This allows a consuming tool to deal predictably with compilers that declare local variables that are never used.
2	Interpretation of source reference column numbers. 0 Source column 1 means 'the last column of the indicated text line'. 1 Source column 0 means 'the last column of the indicated text line', i.e., column 1 means column 1.

ASN3 Default pointer size for module (in MAUs).
ATN1 Version number of tool
ASN4 Year (e.g. 1988)
ASN5 Month (1-12)
ASN6 Day (1-31)
ASN7 Hour (0-23)
ASN8 Minute (0-59)
ASN9 Second (0-59)

3.6.7 R_Label

R_Label codes occur within BB4 and BB6 blocks and signal a procedure exit. The format of the R_Label code is similar to that of the Compiler ID. A dummy NN record precedes the initial ATN record to produce a symbol name index. One ATN record defines that this is a procedure miscellaneous directive. It is followed by one ASN record for the address of the R_Label being defined.

format:
NN record: {$F0}{n1}{0}
ATN record: {$F1}{$CE}{n1}{0}{62}{1}{1}{v1}
ASN record: {$E2}{$CE}{n1}{v1}

where:

$F0	NN record type
n1	Symbol name (NN record) index
0	Symbol name (blank)
$F1CE	ATN Record type
n1	Symbol name index associated with the module misc. ATN.
0	Symbol type index
62	Attribute definition of 62 for procedure misc.
1	Procedure misc. type identification number of 1: R_Label definition
1	Miscellaneous record count
v1	Address of the R_Label being defined
$E2CE	ASN record type
n1	Symbol name (NN record) index
v1	R-label value

3.6.8 Block End (BE)

The BE record extends the IEEE standard and is used in conjunction with a BB record. The BE record for type 4, 6, and 11 BB records are different than others as indicated in the following definitions:

Block End — General

format: {$F9}

where:

$F9	Record type

Block End — for block types 4 and 6

format: {$F9}{n1}

where:

$F9	Record type
n1	Expression defining the ending address of the function (in MAUs)

Block End — for block type 11

format: {$F9}{n1}

where:

$F9	Record type
n1	Expression defining the size in MAUs of the module section

3.7 Data Part

The Data part contains records with both relocatable and fixed data for the module. It is always loaded at the current PC value in the current section. The current section is defined by the SB record and the PC is defined by the ASP record. If no SB record is defined, the current section is specified as 0. If no ASP record is defined, the PC for a section is initially set to the start of the section. P variables can be used in relocation expressions to describe PC-relative relocation.

Note that Section 10.1 of the *IEEE Trial Use Standard* says that the current section is 0 before any SB records are encountered. Note also that Section 10.2 specifies that if no ST record is present for a section, the type is absolute and shall have an assignment to its L variable. Taken together, these statements imply that the example module in 4.1 of the *Standard* is illegal. MRI and HP follow the definition as stated in Section 10.1 of the *IEEE Trial Use Standard*.

3.7.1 Set Current Section (SB)

The SB record defines the current section. SB has no effect on the P variable. †

format: {$E5}{n1}

where:

$E5	Record type
n1	Section index

3.7.2 Set Current PC (ASP)

The ASP record sets a new value for the current PC. An ASP record is required after an SB record to reset the value of the P variable.

format: {$E2}{$D0}{n1}{n2}

where:

$E2D0	Record type
n1	Section index
n2	Expression defining new value (in MAUs)

3.7.3 Load Constant Bytes (LD)

The LD record specifies the number of MAUs to be loaded as constant data.

format: {$ED}{n1}{...}

where:

$ED	Record type
n1	Number of MAUs (1-127)
...	(n1 x MAU size) data bytes

---

† Currently, Microtec's readers process an SB record, then reset the P variable to the start of the section.

3.7.4 Set Forward Reference Value (ASW) †

The ASW record resolves a forward reference from an earlier point at which the corresponding W variable was used. The value of the W variable is the value given in the most recent ASW record effecting that variable. The index must be greater than 31.

format: {$E2}{$D7}{n1}{n2}

where:

$E2D7	Record type
n1	W variable index (>31)
n2	Expression

3.7.5 Set Relocation Base (IR)

The IR record is used to initialize relocation base by designating a letter from G-Z as a ‘‘relocation base’’ and setting it to a specified expression. A field width may also be specified. The relocation letter may subsequently be used in the LR command (below) provided that it is only used to add constant offsets (in MAUs) to the base.

format: {$E3}{l}{n1}[n2]

where:

$E3	Record type
l	Relocation base designator
n1	Expression defining relocation value (in MAUs)
n2	Field width in bits

— If the second expression is omitted, the field width is taken to be the maximum as specified by the AD record (i.e. 32 bits (4 bytes) for the 68000).

† Proposed, not yet implemented.

3.7.6 Load With Relocation (LR)

format: {$E4}{loaditem}...

where:

$E4 Record type
(A) loaditem :={number}{data_byte}...
or
(B) loaditem :={relocation_letter}{number}
or
(C) loaditem :={$BA | $BC | $BE}{expression}[number]{$BB | $BD | $BF}

Form (A) represents a list of constant bytes (not MAUs) where ‘‘number’’ is the number of bytes following. ‘‘number’’ must be ≤ 127.

Form (B) the relocation_letter is specified earlier in an IR record. The number is an offset to be added to the value of the relocation_letter.

Form (C) gives a single expression whose value is to be stored in ‘‘number’’ MAUs of memory. If ‘‘number’’ is not given, it is assumed to be the maximum number of MAUs per address from the AD record (i.e., 4 for the 68000). The beginning and ending codes surrounding the expression indicate truncation checking, and must occur in certain pairings:

{$BA}...{$BB} ‘‘signed’’ (typical for PC relative branches); check if all discarded bits match the most significant bit.
{$BC}...{$BD} ‘‘unsigned;’’ check if discarded bits are all zero.
{$BE}...{$BF} ‘‘either’’ (probably most common); check if discarded bits are all 0’s or all 1’s.

As the grammar shows, loaditems can be repeated in any order.

3.7.7 Load With Translation (LT)

format: {$FA}{loaditem}

where:

$FA Record type
loaditem For a description of loaditem, see *Load With Relocation (LR)*.

3.7.8 Repeat Data (RE)

The RE record specifies data initialization in a compact form.

format: {$F7}{n1}

where:

$F7 Record type
n1 Expression defining number of times to repeat the following LD or LR record data. The IEEE standard has been extended to include repeating LD records. The length of data that can be repeated is limited to 128 bytes.

3.8 Trailer Part

The Trailer part contains the records described below.

3.8.1 Starting Address (ASG)

The ASG record is optional and defines the execution starting address. This expression requires $BE/$BF delimiters.

format: {$E2}{$C7}{n1}

where:

$E2C7	Record type
n1	Expression defining the execution starting address (in MAUs)

3.8.2 Module End (ME)

The ME record defines the end of the module and must be the last record in the module.

format: {$E1}

where: $E1 Module End (ME) Record Type

3.8.3 Checksum Records

The IEEE standard defines two records for managing checksums. The first checks the running checksum (modulo 256) and resets it to zero. The second resets the running total to zero without checking it.

format: {$EE}{n1}
{$EF}

where:

$EE	Record type indicating check and reset
$EF	Record type indicating reset only
n1	Unsigned number in range 0-FF for check (number is always in a single byte)

The last byte added into the total is the EE record type byte. The running total is always set to zero at the start of a module. The MRI/HP definitions include the following additional restrictions:

- No checksum may involve more than 1 part of the 8 parts defined above.
- Any part, other than the Header and Trailer, that contains a checksum must begin with a EF record and end with a EE record.

IEEE - 695 Object Module Format Specification

Revision 4.1

December 21, 1992

CONTENTS

Appendices

(Page intentionally left blank)