UNIVERSITY OF WATERLOO COMPUTER SCIENCE DEPARTMENT
UNIVERSITY OF WATERLOO COMPUTER SCIENCE DEPARTMENT
UNIVERSITY OF WATERLOO COMPUTER SCIENCE DEPARTMENT

The Thoth Linking Loader

Gary R. Sager

October 1977
CS-77-15

The Thoth Linking Loader

Gary R. Sager

Department of Computer Science
University of Waterloo

October 1977

This research was supported by the National Research Council of Canada.

The Thoth Linking Loader

Gary R. Sager

Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada

1. introduction

The Thoth loader and Ale are part of the Thoth language processors. The Thoth loader is a machine-independent, machine-invariant relocating and linking loader. Ale (A Library Editor) is a machine-independent, machine-invariant "editor" which prepares libraries for use by the Thoth loader. The inputs to both the Thoth loader and Ale are *relocatable load code modules* (hereafter referred to as "modules") output by either the Eh compiler (Braga, 1976) or the Thoth assembler (Malcolm and Stafford, 1977). For the loader, this input is a set of directives which are executed to build an output file. The output file contains a program which will run on the target machine. A detailed description of the internals of the Thoth loader is given in Appendix I.

The Eh compiler outputs a module for each Eh external or function. The Thoth assembler outputs a module for each input module. A module has three components of interest for linking: the *module name*, *entry points* and *external references*. For an Eh external or function, the external or function name, module name and single entry point are identical; external references are global symbols referred to in the module. For a module output by the Thoth assembler, the entry points are obtained from the **.ent** directive, external references from the **.ext** directive and the module name is the first entry point defined.

In terms of the Thoth loader directives, a module is delimited by the start module **M** and end module **E** directives; the module name is defined in the **M** directive, entry points are defined by the **G** directives, and external references are made with the **g** directives. Precise definitions of these and other Thoth loader directives are given in Appendices II and III.

2. loading rules

As the Thoth loader reads modules, it builds a directed graph connecting the modules, entry points and external references encountered. Each node of the graph is named by the module it represents; if there are multiple entry points to the module, the entry points are alias names for that module's node in the graph. An arc directed from node A to node B indicates that module A makes an external reference to module or entry point B. If the node does not yet exist, the arc is allowed to "dangle"; missing nodes are called *unresolved external references*. Because the loader cannot determine whether an unresolved reference is for a module or an entry point, it assumes that all unresolved external references are modules, then corrects the graph when the references are resolved. For example, if module A references B and C which are unresolved, there will be dangling arcs from A to B and from A to C; if the module B is later found to contain an entry point C, the two previously mentioned arcs from A are made to point to the newly created node for B (alias C). B and C are then said to be *resolved.*

As the loader takes directives from a file of modules, it will load any module which has not already been resolved. In order for the Thoth loader to load only the modules required to satisfy unresolved external references, the modules must be presented in the form of a library prepared by Ale.

A library is a subtree of the file system; modules are stored one per file directly under the library root node (see Cheriton, et. al., 1977, for a description of the file system). The pathnames of libraries in Thoth are chosen to indicate functionality and machine-dependence; thus, the node */lib/user/ti.10 is the library root node for the TI 990/10 user library and the module for the function .Printf is found in */lib/user/ti.10/.Printf. The library root node file itself contains a symbol table prepared by Ale. The directives used in the symbol table are described in Appendix III. The symbol table is a resume of the **M**, **E**, **G** and **g** directives which appear in the modules comprising the library.

As modules are processed, the Thoth loader builds a directed graph of the program being loaded, as described above; upon encountering a library, the library symbol table is incorporated into the graph being built. Nodes which appear in the loader's graph may match modules in the library symbol table and vice-versa. The nodes in the loader's graph take precedence: only those modules in the library which are unresolved in the graph being built will be incorporated. When a library module is incorporated as a node, all of its outgoing arcs are also incorporated: if these arcs point to nodes already resolved in the loader's graph, no further action is necessary, but arcs pointing to nodes not yet resolved must either incorporate yet more modules from the library or cause a dangling arc in the graph being built. The above discussion is paraphrased by the following algorithm:

1. If a module has already been loaded, the same module occurring in the library will not be considered.
2. For each module in the library not already marked for loading, if an entry point of the module matches an unresolved external reference, the module is marked for loading and its external references are scanned to determine if any additional unresolved external references will result from the loading of the module.
3. Rule 2 is re-applied until no additional modules are marked for loading.
4. Those modules which were marked for loading are subsequently loaded.

order dependencies: Since only one library at a time is processed, it is possible that an external reference in a later library can be resolved from a previous library. In this case, the loader will not be able to resolve the reference unless a later file or library contains a module having the appropriate entry point. If the user specifies all non-library files before libraries, the problem will manifest itself only if there are two libraries and there is a module in the first library which is not required until the second library is scanned.

naming conflicts: Since loading is done on the basis of module names, but external references are resolved by entry points, some possible confusions may result. For modules created by the Eh compiler, the module name and single entry point are the same, but for modules created by the Thoth assembler, the module name corresponds to only one of the entry points. A module whose name conflicts with a module or entry point already resolved will not be loaded. However, if two modules of the same name are encountered in the same file a warning message is issued, as the loader will try to load the second module on the basis of the size of the first. Another problem occurs when a module to be loaded contains an entry point which is not the same as the module name but which conflicts with an entry point or module name already resolved. In this case, if both modules are to be loaded, the first entry point definition will take precedence and a warning message is issued.

Only the assembly language programmer need worry about naming conflicts other than redefining modules within a file, since they cannot be caused by Eh functions or externals.

3. command line

The user can invoke the Thoth loader directly with the command line:

```
*/cmds/eh/uld  pathname
```

where "pathname" denotes a command file whose contents are:

Line 1 (options): 0 or more characters chosen from the following:

- e continue the load even if there are errors.
- d "debug" output: print out file names as they are processed. If the character d appears twice, the module names will also be printed as they are processed.

Line 2 (home directory): specifies the current directory of the user who invoked the loader. This is needed because the loader sets the current node to load libraries, and requires this information to restore it to its original position.

Line 3 (output file): used to specify the file into which the core image will be placed. If this line is blank, no core image will be output.

Line 4 (map file): used to specify the file into which the load map will be written. If this line is blank, no load map will be output.

Lines 5 *and following* (input files): the remaining lines specify files containing loader directives. The first file specifed must be a prologue defining the characteristics of the target machine. The epilogue, if required, must appear last. All inputs will be processed in the order they appear in the command file. The Thoth loader will load modules as described by the rules in the preceding section.

4. load map

The load map contains information concerning the placement of modules and entry points in the logical address space of the target machine. Information is listed in ascending numeric order of placement in memory. For module names, the load map will have a line of the form:

```
name    octal_loc    hex_loc    pathname
```

where octal_loc and hex_loc indicate the target machine address of the named module, and pathname indicates the name of the file or library from which the module was loaded. The latter information is sometimes useful if the user suspects that a module has not been loaded from the proper file or library. For entry points which are not module names, the load map will have a line of the form:

```
name    octal_loc    hex_loc      from: module_name
```

where module_name is the name of the module in which the entry point is defined. In the case of unresolved external references, a line of the form:

```
name    UNRESOLVED, last ref: module_name
```

is output, where module_name is the last module encountered which made a reference to the unresolved external.

The final few lines output in the map are of the form:

```
Rbr# =    octal_num    hex_num
```

these are the final values of the relocation base registers used during loading; they are useful in determining the amount of space taken in each relocation area.

5. relocations

Every symbol resolved by the loader has a value (address) defined as an offset from one of eight *relocation base registers* (**Rbr**'s). The definition is done with the **G** and **T** directives (see Appendix II). This method of defining symbol values serves two purposes. First, different **Rbr**'s are used to correspond to areas of the target machine memory having different addressability; for example, on the Data General NOVA computer, addresses in the range 0-255 can be directly accessed by instructions anywhere in memory, while addresses above this range can only be accessed by indirection or indexing (including use of the program counter as an index register). In this case, one **Rbr** is assigned to each area, and symbol values are defined as offsets from the **Rbr** assigned to the area in which the symbol is to be loaded.

A second, and perhaps more important, application of the **Rbr**'s is to separate the program into segments which reflect functionality; thus, one **Rbr** might be used to define symbol values in code segments and another to define symbol values in data segments. With this arrangement, it would be possible to use some target machines' mapping hardware to protect code from data access and prevent data from being executed. Another possibility is to dedicate an **Rbr** to to define symbols in a segment used only during initialization; after the initialization is complete, the segment can be used as a data segment.

arranging the segments in virtual space

When all symbol values have been defined, it is necessary to arrange the segments in memory. This is done with the **B** directive. The purpose of the **B** directive is to define the order of the segments corresponding to each **Rbr** in memory, and to insure that the base address of each segment will be greater than some minimum value.

The **m** directive is used to compact the output core-image; if there are gaps between the areas defined by the **Rbr**'s, the gaps will appear as null bytes in the core-image file, since there is normally a one-to-one correspondence between the location of bytes in the core-image file and their location in the logical address space of the target machine. The **m** directive undoes this one-to-one correspondence by causing the output bytes to be mapped into a smaller physical space in the core-image file by eliminating the gaps between the areas defined by the **Rbr**'s.

6. bibliography

Cheriton, D. R., M. A. Malcolm, L. S. Melen and G. R. Sager (1977), Thoth, a Portable Real-Time Operating System, University of Waterloo, Computer Science Department, Research Report CS-77-11, October. (presented at 6th SOSP, Purdue University, November, 1977; to appear in the C.A.C.M.)

Braga, R. S. C. (1976), Eh reference manual. University of Waterloo, Computer Science Department, Research Report CS-76-45, November.

Malcolm, M. A. and G. J. Stafford (1977), The Thoth assembler writing kit, University of Waterloo, Computer Science Department, Research Report CS-77-14, October.

Appendix I: the Thoth loader abstract machine

The Thoth loader can be visualized as an abstract machine whose basic unit of information is the byte. The Thoth loader abstract machine executes directives which are presented in the form of byte strings, and outputs a string of bytes which is an executable program for the target machine. When the Thoth loader begins execution, the first set of directives processed must be a prologue which describes the target machine. Thereafter, it is prepared to process files of modules output by the Eh compiler or Thoth assembler, or libraries of modules prepared by Ale.

The Thoth loader makes two passes over the directives; on the first pass, it determines the sizes of all modules and the locations of all external symbols. On the second pass, it performs linking and relocation and outputs the executable program. Note that since all symbols are defined during the first pass, relocation is actually identical to linking; in both cases, the value of a symbol is added into a field in the core image being produced.

components of the abstract machine

The Thoth loader has a number of registers and operational units which are described below. For the reader interested in obtaining a more detailed view of how these are used by the loader, references to the descriptions of directives in Appendix II are given. The components of the abstract machine are:

1. The relocation descriptors (**Rd**[0] through **Rd**[7]): these describe how values (addresses) will appear in multi-byte fields for the relocation operation. They must be defined in the prologue and cannot be redefined. **Rd**[0] serves a special purpose; it describes the values as they appear in the relocation base registers (see below) and in the symbol table. (see **D** and **O** directives)

2. The relocation base registers (**Rbr**[0] through **Rbr**[7]): these give base values for defining the values of symbols in the module being processed. All **Rbr**'s have value zero at the beginning of pass 1. The **Rbr** values are stored as a single word; hence, there is an assumption that the host computer word can contain a target machine address. This restriction is easily removed, with increased cost in symbol table space and execution time. (see **A**, **B**, **I**, **G**, **M**, **R** and **T** directives)

3. The working symbol dictionary (**Wsd**[0] through **Wsd**[255]): provides a shorthand method for accessing entries in the symbol table. (see **g**, **O** and **t** directives)

4. The working register (**Wr**): a 32 byte register used to hold data to be relocated. (see **L** and **O** directives)

5. The symbol tables: the Thoth loader maintains two types of symbol tables. The first contains all external symbols and module names, along with their values. A symbol is defined when a **G** directive setting its value and defining **Rbr** is encountered. Before it is defined, an entry will contain a pointer to the entry for the last module in which it was referenced; this is used to help trace down the source of unresolved external references. Each entry contains a pointer to the entry for the module in which it is defined. Symbols which are module names have the pathname of the file from which they were loaded in their entry.

The second type of symbol table holds entries for symbols local to a module. There is a table of local symbols associated with each module. Local symbol values are defined by the **T** directive.

Values in the symbol table are stored in a single word. The values are defined by adding the two-byte offset in the **G** or **T** directive to the current value of the specified **Rbr**. In all cases, symbol table entries record the **Rbr** which was used to define their value. This information may be used by subsequent **B** directives to adjust symbol table values. (see **B**, **G**, **M** and **T** directives; also refer to the description of the load map)

6. The adder: the Thoth loader has a special addition unit which is used to add values from the symbol table into byte fields in the working register. A detailed description of the operation of the adder is given in the description of the **O** directive.

summary of directives by pass

For reference purposes, we present here listings of the directives which the loader executes on pass 1 and on pass 2. The reader interested in studying the loader on the basis of what actions occur on which pass can use this as a guide to reading the descriptions of directives in Appendix II.

pass 1: **A**, **B**, **b**, **D**, **E**, **G**, **g**, **I**, **M**, **m**, **R** and **T**

pass 2: **E**, **L**, **g**, **M**, **O** and **t**

Appendix II: Thoth loader directives

The general form of a Thoth loader directive is:

c s d...d k

where:

c	is the command byte.
s	is the number of data bytes.
dd...d	are the data bytes.
k	is a checksum (the exclusive-or of c, s, and dd...d).

Individual directives are listed below in alphabetic order of the command byte. Refer to the section describing the structure of the Thoth loader abstract machine for a more complete description of the registers, tables, etc. Since directives may have different actions during passes 1 and 2, the descriptions are divided into explanations of their actions for the two passes. See Appendix III for descriptions of the library symbol table directives.

A align Rbr

A 2 rb m k

where:

rb	is an **Rbr**.
m	is a multiplier to apply to the **Rbr**.
pass 1:	If **Rbr**[rb] is not already an even multiple of m, it is increased to the next multiple of m.
pass 2:	nothing.

B blowup symbols

B 3 rb1 rb2 rb3 k

where:

rb1-3	are **Rbr**'s.
pass 1:	The blowup value is selected as the larger of **Rbr**[rb2] and **Rbr**[rb3]. All symbols whose values are defined from **Rbr**[rb1] will have the blowup value added to their value. **Rbr**[rb1] is incremented by the blowup value.
pass 2:	nothing.
note:	The blowup directive will typically appear only in the epilogue. This directive allows the code generator to place code and/or data into areas defined by several **Rbr**'s, then "blow them up" so that they will fit neatly as a single package of contiguous code and/or data. Without this facility, the size of each area would have to be known before pass 1 of the Thoth loader, so the **Rbr**'s could be given the correct initial values.

b byte definition

b 3 bpc bpb bpw k

where:

bpc	indicates the bytes per target machine-addressable cell.
bpb	indicates the number of bits in a target byte.
bpw	indicates the number of bytes in a target machine Eh word.

pass 1: the Thoth loader sets the appropriate internal constants.

pass 2: nothing.

note: If specified, this directive should appear immediately following the **D** directive for **Rd**[0] in the prologue. These constants default to 1 byte per addressable cell, 8 bits per byte and 2 bytes per Eh word. It is unwise for the uninitiated to specify any bits per byte other than 8, since the Thoth loader enforces 8 bit bytes in many places (this is for easy compatibility with Ale). This is a kludge to make the Thoth loader work for the Honeywell 6000 machines.

The bytes per addressable cell must be set to 2 for word address machines such as the NOVA or the Honeywell Level 6, and should be 4 for the Honeywell 6000. This constant is used when seeking in the output core image file.

D relocation descriptor definition

D s d b t mm...m k

where:

d	is the descriptor number.
b	is a shift indicator (see explanation of **O** directive below).
t	is an indicator of the type of arithmetic to be performed. (0 for forward, 1 for backward)
mm...m	is a mask which defines the field to be relocated

pass 1: b, t and mm...m are copied into **Rd**[d].

pass 2: nothing.

note: **Rd**[0] has special meaning for the Thoth loader: it determines the form of symbol table entry and **Rbr** values. The first directive in the prologue must define **Rd**[0]. It is not possible to redefine the relocation descriptors.

E end of module

E s bb...b k

where:

bb...b	is a byte string used by the compiler.

note: This directive indicates the end of a module. The string of bytes bb...b is not used by the Thoth loader, but is kept by Ale for use in stack bounding, error checking, etc.

G global symbol definition

G s rb aa nn...n k

where:

rb	is an **Rbr**.
aa	is a two-byte offset value relative to **Rbr**[rb].
nn...n	is the symbol name (<= 32 characters).
pass 1:	If the symbol is already in the symbol table and its value defined, a warning message is printed and the original value is left intact. Otherwise, the value of the symbol is defined to be the result of aa+**Rbr**[rb], its defining **Rbr** is set to rb, and the symbol and its value are added to the symbol table.
pass 2:	nothing.

g global symbol reference

g s p nn...n k

where:

p	is a number which is used to refer to this symbol in subsequent **O** directives.
nn...n	is the symbol name.
pass 1:	If the symbol name is not already in the symbol table, an entry is made for it and its value is set to undefined.
pass 2:	The referenced symbol is accessed by placing a pointer to its symbol table entry in **Wsd**[p].

I increment Rbr

I 3 rb aa k

where:

rb	is the **Rbr** to be incremented.
aa	is a two-byte increment value.
pass 1:	**Rbr**[rb] = **Rbr**[rb] + aa
pass 2:	nothing.
note:	The use of a two-byte increment here and in the G and T directives gives a practical limit of 2**16 addressable units per **Rbr** for any single relocatable object module. However, the output absolute module can be arbitrarily large.

L load Wr

L s dd...d k

where:

dd...d	is a string of bytes to be copied into the **Wr**.

pass 1:	nothing.
pass 2:	The string is copied into the **Wr**, starting at the leftmost byte. The contents of Wr will later be output (by the **O** directive) as an absolute module record. Thus, the contents of Wr must resemble an absolute module record; the most convenient form of these records will vary according to the machine and the device used. As an example, we may decide that each absolute record should consist of a two-byte address followed by a byte count and the number of data bytes indicated by the byte count. In this case, the first two bytes of Wr must be converted to an absolute address by the next **O** directive.

M beginning of module

M s rb m nn...n k

where:

rb	is an **Rbr**.
m	is a multiplier to apply to the **Rbr**.
nn...n	is a module name (<= 32 characters).
pass 1:	If **Rbr**[rb] is not already an even multiple of m, it is increased to the next multiple of m; the module name is entered into the symbol table and its value is set to undefined.
pass 2:	The **Wsd** is initialized to zeros to prepare for loading of the new module, and **Wsd**[0] is initialized to point to the symbol table entry for the named module.
note:	The module name will usually be identical to an that of an entry point defined using the **G** directive. Unless the symbol is defined by a **G** directive, its value will be zero and it will have no defining **Rbr**.

m define mapping

m s rb1 ... rbn k

where:

rb1-n	are **Rbr**'s.
pass 1:	The values in the specified **Rbr**'s are used to set up the virtual to real mapping described below.
pass 2:	nothing.
note:	This directive, if used, must appear twice in the epilogue. The first occurrence must precede the **B** directive(s) and the second must follow them. Note that the **m** directive cannot be used without **B** directives. The purpose of this directive is to allow a core image to be created which occupies a large virtual space with unused "holes" which do not appear in the real space in which it is stored. This results in a great saving in file space. The first occurrence of the **m** directive causes the loader to save the information contained in the specified **Rbr**'s as a definition of the real space required by the core

image. The **Rbr**'s must appear in the order in which the regions they represent will appear in virtual space; in particular, this order determines the order the **Rbr**'s must be used in the succeeding **B** directives. For example, if the directive

```
m 4 0 3 1 2
```

were used, then the blowup directives would be

```
B 3 3 0 0
B 3 1 3 3
B 3 2 1 1
```

Here, we have assumed that the selection of a maximum value for blowup is not necessary and have specified the same **Rbr** for the blowup value (refer to the **B** directive). We assume that the value of **Rbr**[0] in this example is the smallest address at which code or data is to be loaded. If this value is 0, the first **B** directive could be omitted. With these blowups, symbols whose defining **Rbr** is 2 will have the highest numbered addresses, those whose defining **Rbr** is 1 will have the next highest and those whose defining **Rbr** is 3 will have the lowest numbered addresses. The second occurrence of the **m** directive must be identical to the first.

O relocate and output Wr

O s cw ... cw k

where:

cw	is a two-byte pair, as follows:
c	has two fields: the first 5 bits (bn) give a byte number in the **Wr**; the remaining 3 bits (rn) is a relocation descriptor number.
w	is an index into the (**Wsd**).
pass 1:	nothing.
pass 2:	For each cw pair, the following relocation algorithm is executed. The symbol table value pointed to by **Wsd**[w] is copied into the AC register, then the AC is right-shifted (as in a division by a power of 2) the number of bits specified in the b field of relocation descriptor **Rd**[rn]. Next, a field F in the **Wr** is isolated by aligning byte bn of Wr with the leftmost byte of the mask bytes from **Rd**[rn]; the 1 bits in the mask bytes define the field F. The value in F is added to the (shifted) value in AC and the result is stored in F.
note:	The loader does not check for overflow conditions, but overflows will not affect data outside the field defined by the **Rd**. Subtractions can be performed by using two's complement representations.

R rb definition

R s rb aa...a k

where:

rb	is the **Rbr** to be set.

aa...a	is the value to be stored into **Rbr**[rb]. The number of bytes must be the same as the number of mask bytes in **Rd**[0] (refer to the **D** directive).
pass 1:	**Rbr**[rb] = aa...a
pass 2:	nothing.

T local symbol definition

T s rb aa nn...n k

where:

rb	is an **Rbr**.
aa	is a two-byte offset value relative to **Rbr**[rb].
nn...n	is the symbol name (<= 7 characters).
pass 1:	The value of the symbol is defined to be the result of aa+**Rbr**[rb], its defining **Rbr** is set to rb, then the symbol and its value are added to a symbol table which is associated with the module in which the definition occurs.
pass 2:	nothing.

t local symbol reference

t s p nn...n k

where:

p	is a number which is used to refer to this symbol in subsequent **O** directives.
nn...n	is the symbol name.
pass 1:	nothing.
pass 2:	The referenced symbol is accessed by placing a pointer to its symbol table entry in **Wsd**[p].

Appendix III: library directives

The following directives appear only in library symbol tables. They are used to provide a summary of information concerning the modules in the library.

F library symbol table header

F 4 ss mm k

where:

ss	is a two-byte number indicating the number of symbols in the library.
mm	is a two-byte number indicating the number of modules in the library.
pass 1:	The symbol table is read in and used to determine which modules from the library will be loaded.
pass 2:	nothing

N library symbol name

N s cc nn...n k

where:

cc	is a two-byte count of the number of times this name appears in subsequent **W**, **X** and **Y** directives.
nn...n	is the symbol name.
pass 1:	The symbol name is used to determine what references can be resolved by loading from this library and what new unresolved references will be created.
pass 2:	nothing
note:	The number of **N** directives is indicated by the value ss in the **F** directive. The **N** directives immediately follow the **F** directive and precede the **W**, **X** and **Y** directives.

W library module name

W s nn pp bb...b k

where:

nn	is a two-byte index into the list of symbols given by the **N** directives. The indicated symbol is the module name.
pp	is an unused two-byte field.
bb...b	is a string of bytes saved from the E directive of the module.
note:	Each **W** directive is always followed by an **X** directive. If the module has entry points other than the module name, the **X** directive will be followed by a **Y** directive; if there is a **Y** directive, it will

include the module name in the list of entry points. The number of **W** directives is indicated by the value mm in the **F** directive. These directives immediately follow the **N** directives.

X external references

X s rr rr ... rr k

where:

rr	is a two-byte index into the **N** directives indicating a symbol to which an external reference is made.
note:	This directive summarizes the **g** directives which appear in the library module.

Y entry points

Y s ee ee ... ee k

where:

ee	is a two-byte index into the **N** directives indicating the names by which the module may be referred.
note:	This directive summarizes the **G** directives which appear in the library module.

Z end of library symbol table

Z 0 Z

where:

	there are no data bytes for this directive.
note:	This directive signals the end of the library symbol table.

Appendix IV: generating load code

Every module must begin with an **M** directive, then **A** directives to align additional **Rbr**'s. After these, **G**, **g**, **T**, **t**, **L** and **O** directives may come in any order, with only the restrictions that **O** directives apply to the previous **L** and the **g** and **t** directives must appear before the **O** directives which use them. A module will typically end by issuing the **I** directives to increment the **Rbr**'s by the amount used in each relocation area. Every module must be terminated by an **E** directive.

The prologue should define all relocation descriptors (**D** directive) which are used. Rd[0] should be defined immediately following the **M** directive in the prologue. Rd[0] is used to define the format of values in the **Rbr**'s and symbol tables. The logical choice for the mask is the value of the largest address in the target machine, right justified in the mask bytes. Next, it should initialize any **Rbr**'s which start from non-zero values (**R** directive). The prologue may also contain other directives to define certain standard information and make reference to externals which must be resolved (i.e., the function Main). The prologue must be terminated by an **E** directive.

The epilogue may be used to define externals whose values will be the final values of the **Rbr**'s; this will allow the program to test how large it is when it begins execution. The **B** directive should be used only in the epilogue and should come after any **G** or **T** directives.

The names of the modules chosen for the prologue and epilogue should not conflict with any other modules. Conflicts may be avoided by using standard Thoth names for them (i.e., starting with a '.') or by using illegal Eh identifiers for their names.

tools: The command Culd may be used to hand-craft the prologue and/or epilogue. Input to the program is best done by preparing a text file using Ted, then redirecting input from the file. The input format is one directive per line, and every input unit is specified as three characters; an input unit which is to be treated as an alphabetic character is specified as two blanks followed by the character, while a byte is specifed as three octal digits. The size of the directive and the checksum should not be specified, as these will be computed by Culd. As an example, the input line

```
M001002  .  N  a  m  e
```

would result in a directive to define the module ".Name" and align **Rbr**[1] to an even boundary. It may be possible to create the epilogue using the Thoth assembler if no **B** or **m** directives are required.

Load code may be inspected using the command Pu.16. This program prints directives with fields appearing in character, string, decimal, hexadecimal or octal form, as appropriate. It also checks where possible that directives are in the form described in Appendices II and III; however, the error checking tends to be rather superficial, since many directives are of variable length and contain arbitrary data.

error checking: The Thoth loader makes little or no attempt to verify the load code it processes. The only error detection done is directed toward discovering user errors rather than compiler errors. However, there has been an attempt to make the code as readable as possible; the compiler writer is encouraged to inspect the code if there are questions not answered in this document, or if the loader appears to be mis-behaving.