To: Distribution
From: M. Weaver
Date: April 23, 1975
Subject: Changes to Object Segment Format

This MTB describes proposed changes to the standard object segment format that will make use of entry parameter descriptors more efficient, allow for the maximum efficiency in prelinking and facilitate certain functions of system initialization. It is related to MTB-169 about the proposed prelinking scheme. Although the changes are not hard to implement, they affect many other parts of the system and many programs will have to be at least recompiled to work with the new format. These changes will also affect some users. This MTB will explain the changes and describe their effects on various system programs. All of these changes can be incorporated into the system without a flag day. MPM documentation describing the changed structures is attached, with a "*" before each new item.

THE CHANGES

There are four basic changes being proposed. These are:

1) optionally moving "static" storage from the linkage section to a section of its own (and hence reformatting the object map),

2) changing the location of the entry bound indicator for gates from an arbitrary convention in the definition section to a standard in the object map,

3) adding the capability for text embedded linkage pairs (primarily for system initialization) and

4) moving entry parameter descriptor pointers from the definition section to the text section.

The object segment format resulting from the first three changes is an alternative to, not a replacement for, the current standard object segment format. Many of these changes will normally be used only by system programs or user programs with special needs. However, the entry sequence change and the new version of the object map are replacements. The changes are described below in more detail.

SEPARATING THE STATIC SECTION

Currently the internal static "section" of an object segment is in the linkage section between the linkage header and the links. The intention was to put all of the object segment's unshared (impure) data in one section and to copy that section at runtime (into the combined linkage segment). Both static and links fall into this category and both are addressed via a pointer to the copy of the linkage section. Now, however, there is a proposed prelinking scheme which snaps the links in some procedures at system initialization time, thus reducing the overhead of dynamic linking and saving pages if the links are shared. To gain the most efficiency from prelinking, only the internal static of these programs is copied at runtime. Because these links and static will reside in two different segments at runtime, they have to be addressed independently. This is all made simpler if static is considered a distinct section, separate from the links. It will have its own entry in the object map (see attached description of the proposed object map) and will normally be located between the linkage and symbol sections. The mechanism for making this section usable by the process is explained in MTB-169. Adding a new section means adding two other features for consistency. One is a new definition class, 4, specifying a segdef in a separate section. The other is a new section value for the self-referencing links, types 1 and 5. The value is 3 and is represented symbolically as *static. In order to avoid changing the symbol section by adding another relocation structure, separate static sections are restricted to having only absolute relocation.

This change is not relevant for most user programs because there is less overhead when the linkage section and static are combined (one template to copy, one pointer register to reference), so having separate static will be optional. Most system programs will be recompiled to have it for prelinking. In addition, programs that know about object segment formats such as the compilers, the binder, the linker, stu_, object_info_ and all its callers, etc., must be modified to handle the separate static. More detailed information about the changes involved is given later in this MTB.

MOVING THE ENTRY BOUND INDICATOR

Having an indicator within a gate object segment of where the entry transfer vector ends facilitates manipulation of gates. Because all gates are in alm, only that language is affected. Currently, by convention, one includes the segdef "tv_end" at the end of the transfer vector, which causes a definition to be constructed. It would be much less awkward to have this value in the new version of the object map now that there is one anyway. There is a new alm pseudo-op, entrybound, to delimit the transfer vector. Non hardcore gates do not yet have their entry bounds set in their branches, which means that they do not use tv_end and are not affected yet by this change. Hardcore gates must continue to use tv_end until the MST generator is changed to retain object maps, since the entry bounds used are picked up from the "object" segments during system initialization.

TEXT EMBEDDED LINKS

With this change a threaded list of linkage pairs would be allowed in the text section. Each pair would look like a normal link except that the first 18 bits in each pair would point to the next pair instead of to the linkage section header, with the first pair pointed to by the object map. The original motivation behind this proposal was to simplify system initialization. There are already several hardcore programs that have pointers in the text filled in by special initialization programs. This scheme would allow the pointers to be filled in by the system prelinking mechanism. Moreover, with prelinking of the user ring system as well, text embedded links could be used in other programs (such as PL/I programs) to eliminate linkage sections altogether and to prevent unlinking. For now, however, only alm will produce them. These links would comprise a logically independent section since they are pointed to by the object map. The reason for not having them actually be together as a separate section is so that each link can be placed in the text near where it is used; this would minimize paging. These links would be snapped at system initialization time. It is not intended that the linker be changed to handle this unusual format; it could be done fairly easily but the object segments involved would have to be either modifiable or copied on write. Although in practice these links would probably not be faulted on, they will start out as linkfaults (fault tag 2) in case it is decided in the future to have the linker handle them in some way, for example to snap links left unsnapped by the prelinker. It is clear that besides the convenience, allowing text embedded links would open up several research opportunities.

MOVING ENTRY DESCRIPTOR POINTERS

The current standard object segment format specifies that the pointers to an entry's parameter descriptors, if they exist, be appended to the definition. At the time this was designed, it was not clear exactly how and when they would be used. So far they are used only by get_entry_arg_descs_, which is called by trace and trace_stack. Now, however, the command processor is being changed to look at the entry parameter descriptors and it has only a pointer to the entry sequence in the text. It does not want to go to the trouble of looking at the definition, especially since the parameter descriptor pointers are logically more a part of the entry sequence than of the definition. The command processor will be the heaviest user of the descriptor pointers so it is worthwhile to optimize what it does. The expense of looking at the definition involves at least touching an extra page in many cases, since the definition would otherwise not be paged in after the first invocation (or at all under prelinking for system commands), and getting a pointer to the definition section from the linkage section copy whose location is in the LOT. Moreover with the linker in the user ring, the ready-made definition section pointer cannot be guaranteed. Thus, to be safe, the command processor would have to call into ring 0 to get the bit count and then call object_info_ just to get a pointer to the definition section. Getting everything from the entry sequence is clearly preferable.

The other users and potential users of the descriptor pointers include get_entry_arg_descs_, the binder and runtime parameter checking, which are not as heavily used as the command processor (note that with prelinking, parameter checking will not be performed for system-called subroutines except perhaps at prelinking time) and which will not have trouble with the new format. They either start out with the entry pointer or can get it very quickly from the definition. Paging should not be significantly increased because of this change if at all.

The actual changes being proposed are to add some more entry flags to the word containing the definition offset, to put a relative pointer to the descriptor pointer array in the word preceding the flags, and to move the pointer array from the definition to the text section. The revised MPM description is attached. The flags are defined so that one can determine the exact parameter setup from the entry sequence. In order to tell the PL/I compiler when to turn on the variable flag, a new option, variable, will be available for the procedure and entry statements.

EFFECTS OF OBJECT SEGMENT CHANGES

Listed below are most of the system routines that have to be changed to handle the new object segment format, along with an explanation of the changes and how they can be made compatibly.

<u>alm changes</u>

The assembler needs several new pseudo-ops and extensions to some old ones. The new ones are:

1) link_in_text <segment_name>|[entryname]+exp,mod
indicates that a text-embedded link should be inserted in the text at the current location.

2) entrybound
indicates that the entrybound field in the object map should be set to the current location.

The extensions are:

1) The join pseudo-op will also accept /static/. If and only if this is specified a separate static section is created. Specifying both join /link/ and join /static/ in the same program is not allowed. Definitions for segdefs defined in static are given the definition class 4.

2) <*static> will be accepted as a legal segment name both in the segment field of addresses and by the link pseudo-op. It will cause the generation of a link of type 1 or 5 to the static section. This specification is also not allowed in the same program with a join /link/ statement.

3) The push pseudo-op will cause a transfer to a new push operator if a separate static section is used so that the static pointer can be set. alm programmers who use separate static must take care to reference their own static locations via a static pointer, which the push operator will return at pr6|28, instead of via the linkage pointer, pr4. If desired, a command can be provided to check for references to static via pr4.

PL/I Changes

The PL/I compiler will create a separate static section in the object segment if the -separate_static (-ss) control argument is given. In this case, internal static variables are referenced via the static pointer, obtained by the entry operator, instead of via the linkage pointer. (Most users will want to keep their static and linkage combined to reduce overhead.) The entry parameter descriptor pointers will be moved from the definition to the entry sequence in the text section.

pl1_operators_ changes

Alternate operators will be provided for each PL/I entry operator which will be identical except that the static pointer will be obtained and stored at pr6|28. The current entry operators will continue to be used when a separate static section is not generated. The alm entry operator will be changed to always obtain the static pointer and store it at pr6|28. A new alm operator would have meant another operator pointer the the cramped stack header.

binder changes

The binder must be able to handle all three object segment formats, including the one used before the current standard format. The conversion of the system for prelinking will be simplified if separate static components can be bound with combined static components, since it is unlikely that all components of some bound segments will be replaced simultaneously. However, a bound segment will not have a separate static section unless each component has one; to have internal static for some components in the linkage section and for others in the static section would add unnecessary complexity and the linkage section would not be shared anyway. When there are mixed components, the separate static sections will be put in the linkage section and relocated with respect to the beginning of the linkage section; no record will be kept in the bound segment of which static sections were originally separate. However if all of the components that have static sections have separate ones, the bound segment itself will have a separate static section.

The binder's ability to create separate static sections has one major effect that may cause some bound segments to keep their static in the linkage section. That is that the binder cannot prelink to segdefs in a separate static. Currently when one component references segdefs in another component's static, the binder takes advantage of the fact that the static is combined with the linkage section by changing only an offset and an indirection to convert an instruction referencing through a link in one component to one referencing the static of another component. When the static section is no longer combined with the linkage section, that trick will no longer work. There is no plan to reserve a pointer register for the static pointer and to ensure that it is valid before every link reference so that the binder can substitute it for the linkage section pointer. There are thus two choices for bound segments to be prelinked with the system that have segdefs in static. These are:

1) Do not recompile with separate static. All references to the bound segment's static segdefs from within the bound segment will continue to be direct references (using pr4). However, the links cannot be shared so each user of the bound segment will have his/her own copy of the entire linkage section. Of course the links will still be presnapped.

2) Recompile all components with separate static so that the bound segment has a separate static section. All references to the bound segment's static segdefs from within the bound segment will be indirect references through links. However the links will be presnapped and shared.

<u>object_info_</u>

The structure that object_info_ fills in must be changed to reflect the new object map information. See the attached writeup for a description of the new structure; the additional items are starred. It is important to be able to handle both structure declarations since the callers of object_info_ cannot be changed all at once. To distinguish the structures, object_info_ will rely on the version number which, unlike the other structure items, must be filled in by the <u>caller</u>. The version described here is number 2. Because many callers do not yet fill in the number, any other number is considered version 1 for the time being and in that case the current structure will be filled in. Unless some of the new items are relevant, there is no way to tell from the structure whether the object segment has a version 1 or a version 2 object map, but that knowledge should not be necessary.

A couple of the items could use further explanation. The static pointer is always meaningful. If the segment does not have a separate section, the static pointer points to the actual beginning of the static region within the linkage section. If there is no static section, i.e., it is zero length, the static pointer is null. call_delimiter has been renamed entry_bound to correspond with the object map. It is not filled in unless it is nonzero in the object map since object_info_$brief should not search the definitions of all alm segments for tv_end, which only hardcore gates have.

There is no way to make the desired include file changes compatibly for everyone. Currently there are two include files with the same structure name, one automatic and one based. It is better to have one include file and for those who want a different storage class to use the "like" attribute; incomplete structures omitting level 1 are to be avoided. So that no one need change references to the structure, the best solution seems to be to change the structure name in the include file and to require everyone to add a structure declaration for the old name using the "like" attribute. The attached MPM declaration is the same as the new include file. (Automatic variables don't get allocated if they are not referenced unless a table is produced.)

callers of object info

There are about 25 system procedures that call object_info_. These all have to be changed to use the new structure. Many will need little more than recompilation with the new object_info_.incl.pl1; all except for the binder and the linker should need only minor changes.

decode definition

Besides changes in calling object_info_, decode_definition_ needs to recognize the new definition class 4 for static.

form link info

Besides changes in calling object_info_, form_link_info_ (print_link_info) needs to know about both the new definition class and the new value (*static) for self-referencing links.

command processor

The command processor will look only in the text for entry parameter descriptors. If it finds them, it will create descriptors for the argument list it builds if all the parameters are character strings.

get_entry_arg_descs_

This must be changed to look at the entry sequence for the parameter descriptor pointers. If they are not found in the text, the entry's definition must be checked.

EFFECTS OF HAVING SEPARATE STATIC AT RUNTIME

The procedures listed below are not interested in the object segment changes themselves as much as in the effects of having static separate from the linkage section at runtime and accessed via the ISOT (Internal Static Offset Table) rather than the LOT (Linkage Offset Table).

linker

The linker will have the added responsibility of managing the ISOT. Of course it also has to know about separate static, class 4 definitions and self-referencing links to *static.

stu_

stu_ will sometimes need the static pointer to access a segment's internal static variables. It seems preferable for stu_ to obtain the pointer itself than to add a new entrypoint for each of the five entrypoints that might be interested in it. This would also save changing the 10 or so callers. To isolate the cases where a separate pointer is needed, a new code will be used in the symbol table to indicate that a variable is in separate static. When the static pointer is needed, stu_ will obtain it from the ISOT. The installation of this must be carefully arranged to occur after the ISOT management is installed but before there are any object segments with separate static.

debug

debug needs a new segment ID, &i, for internal static. The offset used should be the same as that in the listing, so &i is equivalent to &l for static sections that are not separate.

trace_stack

The change to get_entry_arg_descs_ has already been described. interpret_ptr_ must call a different routine than is_cls_ to determine whether the pointer points to someone's internal static.

bound_debug_util_ procedures

Some of these, particularly find_ls_owner_ and is_cls_, need to look at the ISOT as well as the LOT. Perhaps there should be a new procedure, is_static_, for use by interpret_ptr_. The programs that know about operators have to be updated.

dump_ls

This command must merge the ISOT with the LOT to be able to dump the combined linkage section continuously. The output will change slightly to accomodate separate static sections.

dump_static

This suggested command would dump only static sections for users not interested in links. It may be more desirable when most links have been prelinked.

print_linkage_usage

Like dump_ls, this command must merge the LOT and ISOT and the output may need to be modified slightly.

THE STRUCTURE OF THE OBJECT MAP

The object map contains information which allows the various sections of an object segment to be located. The map itself can be located immediately before or immediately after any one of the five sections. Translators normally place it immediately after the symbol section. The last word of the segment must contain a left-justified 18-bit pointer (relative to the base of the object segment) to the object map. The object map has the following format:

```
     declare 1 object_map aligned,
                2 decl_vers fixed bin init(2),
                2 identifier char(8) aligned,
                2 text_relp bit(18) unaligned,
                2 text_length bit(18) unaligned,
                2 def_relp bit(18) unaligned,
                2 def_length bit(18) unaligned,
                2 link_relp bit(18) unaligned,
                2 link_length bit(18) unaligned,
               *2 static_relp bit(18) unaligned,
               *2 static_length bit(18) unaligned,
                2 symb_relp bit(18) unaligned,
                2 symb_length bit(18) unaligned,
                2 bmap_relp bit(18) unaligned,
                2 bmap_length bit(18) unaligned,
               *2 entry_bound bit(18) unaligned,
               *2 text_link_relp bit(18) unaligned,
                2 format aligned,
                  3 bound bit(1) unaligned,
                  3 relocatable bit(1) unaligned,
                  3 procedure bit(1) unaligned,
                  3 standard bit(1) unaligned,
                 *3 separate_static bit(1) unaligned,
                 *3 links_in_text bit(1) unaligned,
                  3 unused bit(30) unaligned;
```

1. decl_vers — is the version number of the structure.

2. identifier — is the constant "obj_map".

3. text_relp — is a pointer (relative to the base of the object segment) to the base of the text section.

4. text_length — is the length (in words) of the text section.

5. def_relp — is a pointer (relative to the base of the object segment) to the base of the definition section.

6.	def_length	is the length (in words) of the definition section.
7.	link_relp	is a pointer (relative to the base of the object segment) to the base of the linkage section.
8.	link_length	is the length (in words) of the linkage section.
9.	static_relp	is a pointer (relative to the base of the object segment) to the base of the static section.
10.	static_length	is the length (in words) of the static section.
11.	symb_relp	is a pointer (relative to the base of the object segment) to the base of the symbol section.
12.	symb_length	is the length (in words) of the symbol section.
13.	bmap_relp	is a pointer (relative to the base of the object segment) to the base of the break map section.
14.	bmap_length	is the length (in words) of the break map section.
15.	entry_bound	is the offset of the end of the entry transfer vector if the object segment is to be a gate.
16.	text_link_relp	is the offset of the first text-embedded link if item links_in_text = "1"b.
17.	bound	is "1"b if the object segment is a bound segment.
18.	relocatable	is "1"b if the object segment is relocatable; that is, if it contains relocation information. This information (if present) must be stored in the segment's first symbol block. See the MPM Subsystem Writers' Guide section, The Structure of the Symbol Section.
19.	procedure	is "1"b if this is an executable object segment.
20.	standard	is "1"b if the object segment is in standard format.

21. separate_static is "1"b if the static section is separate from the linkage section.

22. links_in_text is "1"b if the object segment contains text-embedded links.

23. unused is reserved for future use and must be "0"b.

THE STRUCTURE OF THE TEXT SECTION

The text section is basically unstructured, containing the machine language representation of some symbolic language algorithm and/or pure data items. Its length must be an even number of words.

Two items which can appear within the text section have standard formats; namely the entry sequence and the gate segment entry point transfer vector.

The Entry Sequence

There must be a standard entry sequence for every externally accessible procedure entry point in an object segment. It has the following format (the two structures are independent but are normally contiguous).

```
declare 1 parm_desc_ptrs aligned,
         *2 n_args bit(18) unaligned,
         *2 descriptor_relp(n_args) bit(18) unaligned,

declare 1 entry_sequence aligned,
         *2 descr_relp_offset bit(18) unaligned,
         *2 reserved bit(18) unaligned,
          2 def_relp bit(18) unaligned,
          2 flags unaligned,
            3 basic_indicator bit(1) unaligned,
           *3 revision_1 bit(1) unaligned,
           *3 has_descriptors bit(1) unaligned,
           *3 variable bit(1) unaligned,
           *3 function bit(1) unaligned,
            3 pad bit(13) unaligned,
          2 code_sequence(n) bit(36) aligned;
```

1. n_args — is the number of arguments expected by this external entrypoint. This item is valid only if the flag has_descriptors = "1"b.

2. descriptor_relp — is an array of pointers (relative to the base of the text section) which point to the descriptors of the corresponding entrypoint parameters. This item is valid only if the flag has_descriptors = "1"b.

3. descr_relp_offset is the offset (relative to the base of the text section) of the n_args item. This item is valid only if the flag has_descriptors = "1"b.

4. reserved — is reserved for future use and must be "0"b.

The preceding items are optional.

5. def_relp — is a pointer (relative to the base of the definition section) to the definition (see below) of this entrypoint. Thus, given a pointer to an entrypoint, it is possible to reconstruct its symbolic name for purposes such as diagnostics or debugging.

6. basic_indicator — is "1"b if this is the entrypoint of a BASIC program.

7. revision_1 — is "1"b if all of the entry's parameter descriptor information is with the entry sequence, i.e., if none is in the definition.

8. has_descriptors — is "1"b if the entry has parameter descriptors; i.e., items n_args, descriptor_relp and descr_relp_offset contain valid information.

9. variable — is "1"b if the entry expects arguments but the number and types are variable.

10. function — is "1"b if this is a function entry, i.e., if the last parameter is to be returned by this entry.

11. pad — is reserved for future use and must be "0"b.

12. code_sequence — is any sequence of machine instructions satisfying Multics standard calling conventions. See the MPM Subsystem Writers' Guide section, Subroutine Calling Sequences.

Note that the value (i.e., offset within the text section) of the entry point corresponds to the address of the code_sequence item. (The value is stored in the formal definition of the entry point. See the MPM Subsystem Writers' Guide section, The Structure of the Definition Section.) Thus, if entry_offset is the value of the entry point ent1 then the def_relp item pointing to the definition for ent1 is located at word (entry_offset - 1).

The Gate Segment Entry Point Transfer Vector

For reasons of protection, control must not be passed to a gate procedure at other than its defined entry points. To enforce this restriction, the first n words of a gate segment with n entry points must be an entry point transfer vector. That is, the kth word. ($0 \leq k \leq n-1$) must be a transfer instruction to the kth entry point (i.e. a transfer to the code_sequence item of a standard entry sequence as described above). In this case, the value of the kth entry point is the offset of the kth transfer instruction (i.e. word k of the segment) rather than the offset of the code_sequence item of the kth entry point.

To ensure that only these entries can be used, the hardware enforced entrybound of the gate segment must be set so that the segment can be entered only at the first n locations.

Name: object_info_

This procedure returns structural and identifying information extracted from an object segment. It has three entry points returning progressively larger increments of information. All three entry points have identical calling sequences, the only distinction being the amount of information returned in the info structure described below.

Entry: object_info_$brief

This entry only returns the structural information necessary in order to be able to locate the object's four sections.

Usage

```
declare object_info_$brief entry (ptr, fixed bin(24), ptr,
        fixed bin(35));

call object_info_$brief (segp, bc, infop, code);
```

1. segp — is a pointer to the base of the object segment. (Input)

2. bc — is the bit count of the object segment. (Input)

3. infop — is a pointer to the info structure in which the object information is returned. (Input)

4. code — is a standard Multics status code. (Output)

Entry: object_info_$display

This entry returns, in addition to the $brief information, all the identifying data required by certain object display commands, such as print_link_info.

Usage

```
     declare object_info_$display entry (ptr, fixed bin(24), ptr,
             fixed bin(35));

     call object_info_$display (segp, bc, infop, code);
```

1-4)   as above. (Input/Output)

Entry: object_info_$long

This entry returns, in addition to the $brief and $display information, the data required by the Multics binder.

Usage

```
     declare object_info_$long entry (ptr, fixed bin(24), ptr,
             fixed bin(35));

     call object_info_$long (segp, bc, infop, code);
```

1-4)   as above. (Input/Output)

Notes

A description of the information structure follows. A declaration of it is available in object_info_.incl.pl1, which is a standard Multics include file.

```
     declare 1 obj_info aligned,
               2 version_number fixed bin,
               2 textp ptr,
               2 defp ptr,
               2 linkp ptr,
              *2 statp ptr,
               2 symbp ptr,
```

```
          2 bmapp ptr,
          2 tlng fixed bin,
          2 dlng fixed bin,
          2 llng fixed bin,
         *2 ilng fixed bin,
          2 slng fixed bin,
          2 blng fixed bin,
          2 format,
            3 old_format bit(1) unaligned,
            3 bound bit(1) unaligned,
            3 relocatable bit(1) unaligned,
            3 procedure bit(1) unaligned,
            3 standard bit(1) unaligned,
            3 gate bit(1) unaligned,
           *3 separate_static bit(1) unaligned,
           *3 links_in_text bit(1) unaligned,
            3 pad bit(28) unaligned,
          2 entry_bound fixed bin,
         *2 textlinkp ptr,

/*This is the limit of the $brief info structure.*/

          2 compiler char(8) aligned,
          2 compile_time fixed bin(71),
          2 userid char(32) aligned,
          2 cvers aligned,
            3 offset bit(18) unaligned,
            3 length bit(18) unaligned,
          2 comment,
            3 offset bit(18) unaliged,
            3 length bit(18) unaligned,
          2 source_map fixed bin,

/*This is the limit of the $display info structure.*/

          2 rel_text ptr,
          2 rel_def ptr,
          2 rel_link ptr,
          2 rel_symbol ptr,
          2 text_boundary fixed bin,
          2 static_boundary fixed bin,
          2 default_truncate fixed bin,
          2 optional_truncate fixed bin;
```

/*This is the limit of the $long info structure.*/

1. version_number — is the version number of the structure (currently = 2). This value is input.

2. textp — is a pointer to the base of the text section.

3. defp — is a pointer to the base of the definition section.

4. linkp — is a pointer to the base of the linkage section.

5. statp — is a pointer to the base of the static section.

6. symbp — is a pointer to the base of the symbol section.

7. bmapp — is a pointer to the break map.

8. tlng — is the length (in words) of the text section.

9. dlng — is the length (in words) of the definition section.

10. llng — is the length (in words) of the linkage section.

11. ilng — is the length (in words) of the static section.

12. slng — is the length (in words) of the symbol section.

13. blng — is the length (in words) of the break map.

14. old_format — is "1"b if this segment is in the old format; otherwise it is "0"b.

15. bound — is "1"b if this is a bound segment; otherwise it is "0"b.

16. relocatable — is "1"b if the object is relocatable; otherwise it is "0"b.

17. procedure	is "1"b if it is a procedure; is "0"b if it is nonexecutable data.
18. standard	is "1"b if this is a standard object segment; otherwise it is "0"b.
19. gate	is "1"b if this is a procedure generated in the gate format; otherwise it is "0"b.
20. separate_static	is "1"b if the static section is separate from the linkage section; otherwise it is "0"b;
21. links_in_text	is "1"b if this object segment contains text-embedded links; otherwise it is "0"b.
22. pad	is currently unused.
23. entry_bound	is the call delimiter value if this is a gate procedure.
24. textlinkp	is a pointer to the first text-embedded link if links_in_text = "1"b.

This is the limit of the $brief info structure.

25. compiler	is the name of the compiler which generated this object segment.
26. compile_time	is the date and time this object was generated.
27. userid	is the access id of the user in whose behalf this object was generated.
28. cvers.offset	is the offset (in words), relative to the base of the symbol section, of the aligned variable length character string which describes the compiler version used.
29. cvers.length	is the length (in characters) of the compiler version string.
30. comment.offset	is the offset (in words), relative to the base of the symbol section, of the

aligned variable length character string containing some compiler generated comment.

31. comment.length — is the length (in characters) of the comment string.

32. source_map — is the offset (relative to the base of the symbol section) of the source map.

This is the limit of the $display info structure.

33. rel_text — is a pointer to the object's text section relocation information.

34. rel_def — is a pointer to the object's definition section relocation information.

35. rel_link — is a pointer to the object's linkage section relocation information.

36. rel_symbol — is a pointer to the object's symbol section relocation information.

37. text_boundary — partially defines the beginning address of the text section. The text must begin on an integral multiple of some number, e.g., 0 mod 2, 0 mod 64; this is that number.

38. static_boundary — is analogous to text_boundary for internal static.

39. default_truncate — is the offset (in words), relative to the base of the symbol section, starting from which the symbol section can be truncated to remove nonessential information (e.g., relocation information).

40. optional_truncate — is the offset (in words), relative to the base of the symbol section, starting from which the symbol section can be truncated to remove unwanted information (e.g., the compiler symbol tree).

This is the limit of the $long info structure.