SETL Data Structures

by

Stefan M. Freudenberger

SETL Newsletter Number 189B
Courant Institute of Mathematical Sciences
New York University

May 15, 1980

Revised October 23, 1982

This newsletter describes the data structures which are used for the run-time systems of both the interpreted and compiled versions of the SETL 2.1 system. It replaces SETL Newsletter 189 on the SETL Data Structures by Robert B. K. Dewar, Art Grand, Edmond Schonberg, and Jacob T. Schwartz.

Most of the changes to the previous newsletter are in the details, rather then in the general design of the data structure. In the course of the implementation, however, a sufficient number of differences developed to make this revision necessary.

Table of Contents

1.0 FORM OF STORAGE

As in the previous system, storage is arranged as a contiguous sequence of "words". Each SETL word may be composed of one or more machine words depending on the machine word size.

The SETL word (henceforth simply called a word) is capable of holding a pointer field plus 7 additional bits. If more bits are available, they can be made use of in various situations.

There are basically two word formats:

value specifier: A one word quantity used to hold (directly or indirectly) a SETL data value. These correspond to the root words of the old system.

data word: In some cases, value specifiers are used to point to data blocks consisting of one or more data words in a data type dependent format.

1.1 Value Specifiers

A value specifies a data value in the SETL system and has the following uniform fields:

is om: Flags undefined values (Omega values, see section 2.7.)

type: A field giving the type code of the value. This field must be at least 4 bits long, but may be longer if convenient for efficient access. It contains a system constant value (whose name is of the form T xxx) which indicates the type.

value: This field contains either the data value or a pointer to a data block in the heap containing the value. The type code (in the type field) indicates which of these two forms is used and the exact significance of the value field.

is shared: A flag which is set in a specifier to indicate that the value referred to is shared and that the value must be copied before it is modified.

is_multi: A flag used in map formats to indicate that an entry is for a range set (rather than a single range value).

The IS_OM, TYPE and VALUE fields are adjacent and in that order. The following composite fields are defined:

otype: is_om + type

tvalue: type + value

otvalue: is_om + type + value

1.2 Data Words

Certain data values (large primitive values and all composite values) cannot be represented by a single value specifier. These values are represented by one or more data words which are pointed to by the value field of the specifier. Such a collection of data words is called a data block.

Each data block is regarded as an entity by the garbage collector and there is a standard header at the beginning of each data block for use by the garbage collector. This header has the following fields:

htype: A unique code for the type of data block (H xxx)

hlink: Pointer field for use by the garbage collector

hsize: Field for use by the garbage collector in dead blocks to record the size of the next live block.

As with all fields in data blocks, the position of the fields (word and bit offset) is a function of the implementation, subject to the requirement that all pointers have the same position in a word (i.e. they correspond to the standard field STD PTR in each LITTLE word).

HLINK is used to build back chains by the garbage collector. See garbage collector description for full details. Since the garbage collector traces pointers from the symbol table, HLINK may contain any value outside the range (SYM_ORG..H ORG), and the garbage collector preserves this value. Note that this peculiar range is based on the fact that the symbol table precedes the heap in physical memory.

During the garbage collection process, an additional field called HSIZE is used in dead blocks only. Since this field is not

needed in active blocks, it may overlap other fields (but there must be room for it).

The format of the remaining data words in the data block is dependent on the data type involved, and is interpreted as a function of the specifier's TYPE field and the data block's HTYPE field.

If more than one data word is involved, then there is typically additional header information which (among other information) gives or implies the length of the block.

The detailed format of data words is explained data type by data type in sections 2.0 through 6.0.

2.0 TYPED PRIMITIVE DATA

This section contains details on the format of primitive data items which do not contain component values (i.e. all types except sets and tuples).

Every data value in the SETL system is represented by a value specifier (often just called a specifier). In some cases, the data value can be completely contained in the specifier. These values are called short items. In other cases, the value field of the specifier contains a pointer to data words which describe the data value. Such values are called long items.

Note that only the type and value fields contain or represent the data value itself. All the other fields in a specifier are independent of the data value and their use depends on the context in which the specifier appears.

If there are short and long formats for the same data type, then the value is represented in the short form whenever possible.

2.1 Integers

An integer in SETL is a signed integral value with no limit on the magnitude other than that imposed by memory constraints. There are two formats for integers. called short integer and long integer.

2.1.1 Short Integers -

Short integers range from 0..MAXSI, where MAXSI is the largest integer which will fit into the value field (i.e. all bits of the value field set on). The specifier has:

type:	T INT
value:	unsigned integer value

The type field value (i.e. T INT) must be zero. This particular value is significant since it allows for rapid coding of the integer addition function.

The integer value ranges from 0 to the implementation constant MAXSI. Only small positive integers can be stored in this manner. Large and negative integers are stored in long integer format.

2.1.2 Long Integers -

The specifier for a long integer has:

type:	T LINT
value:	Pointer to an integer data block

An integer date block has the following fields:

htype:	H_LINT
hlink:	Not used (0)
li_nwords:	Number of words in the integer data block including the header.

The length of the header is given by the system constant HL_LINT.

The remaining words of an integer data block contain a signed integer in a format which is convenient to the particular machine and implementation. The details of this format are of significance only to the long integer routines and are described there.

The current CIMS implementation places an implementation restriction on the size of an integer: the maximum signed integer has to fit into one SETL word. It should be pointed out, however, that this is an implementation restriction and not part of the language definition.

2.2 Strings

A string value in SETL is an arbitrary length sequence of characters. The exact range of possible characters depends on the implementation environment, but should include upper/lower case letters if possible. There are two formats for string values:

2.2.1 Short Strings -

Short string format can represent strings from zero characters (the null string) to strings of length SC_MAX (an implementation dependent system constant). The specifier for a short string value has:

type:	T STRING
value:	subdivided into two fields as follows:
sc_nchars:	number of characters in string
sc_string:	string characters

The characters are stored left justified, right filled. The fill character is binary zero. The length of a single character in bits is given by the system constant CHSIZ.

N.B. While the current system includes short character strings, they are nowhere generated or used. Instead, all strings are represented as long character strings. Code found in the current run-time library operating on short strings is out-of-date with the current implementation of SETL.

2.2.2 Long Strings -

There are two methods of storing long string values. The choice between these methods depends on the implementation word capacity. The library code has an assembly switch (SSI) which determines the choice for a given machine.

2.2.2.1 Direct Format Long Strings -

This format corresponds to SSI being set off and is used if a string descriptor as described here can fit into the value field of a specifier. The specifier for a direct format long string value has:

type: T ISTRING

value: string descriptor, consisting of:

lc_len: Number of characters in the string

lc_ofs: Offset to first character in string data block

lc_ptr: Pointer to long string data block

The data block which contains the actual string characters is called a long string data block and has the following fields:

htype: H_LSTRING

hlink: Not used (0)

lc_nwords: Number of words in data block, including header

The length of the header is given by the system constant HL_LCHARS.

The characters are stored in the remaining area of the block in a format which is implementation dependent and described by the coding of the string manipulation routines. Note that the string descriptor may reference only a substring of the string contained in a long string data block.

2.2.2.2 Indirect Format Long Strings -

This format corresponds to the SSI switch being set and is used if the three fields needed to describe a string cannot fit into the value field of a specifier. The specifier for an indirect format long string value has:

type: T ISTRING

value: Pointer to string descriptor block

The string descriptor is a pointer to an indirect string data block which has the following format:

htype: H_ISTRING

hlink: Not used (0)

lc_len: Number of characters in string

lc_ofs: Offset to first character in string data block

lc_ptr: Pointer to long string data block

The length of the string descriptor block is given by the system constant HL_IC. There are no additional data words in a string descriptor block.

The data block which contains the actual string characters is called a long string data block and has the following fields:

htype: H_LSTRING

hlink: Not used (0)

lc_nwords: Number of words in data block, including header

The length of the header is given by the system constant HL_LCHARS.

The characters are stored in the remaining area of the block in a format which is implementation dependent and described by the coding of the string manipulation routines. Note that the indirect string data block may reference only a substring of the string contained in a long string data block.

2.3 Reals

There is no provision for short real values since it is assumed that the value field is too short to contain a meaningful real value. However, it is assumed that a full SETL word will hold a real value.

The specifier for a long real value has:

type: T REAL

value: Pointer to real data block

A real data block contains the following fields:

htype: H_REAL

hlink: Not used (0)

rval: Real value

The size of a real data block header is given by the system constant HL_REAL. Since a real data block header is always followed by exactly one data word, the size of a real data block is given by the system constant REAL_NW.

2.4 Procedures

The specifier for a procedure has the following fields:

type: T PROC

value: Code pointer for the routine code

The format of the code pointer depends on the form of the executing program. For the interpretive version, it is a pointer to a code data block. For the compiled version, it is an external name whose absolute address will be suplied by the target machine's linkage editor.

Note that SETL does not distinguish between functions and subroutines. Rather, every routine returns a value, in the case of a subroutine this value is the undefined value omega, and is discarded.

A code data block in the interpretive version has the following fields:

htype: H_CODE

hlink: Not used (0)

codenw: length of code data block, including header

The length of the code data block header is given by the system constant HL_CODE.

The remaining words in a code data block consists of interpretable instructions. The length of one instruction is given by the system constant INST_NW.

2.5 Atoms

An atom in SETL represents a unique value which can be compared for equality but not otherwise manipulated. There are two forms for atoms, called short and long atoms.

2.5.1 Short Atoms -

The specifier for a short atom value has:

type: T ATOM

value: an integer in the range 1..MAXSI-1, uniquely identifying the atom

2.5.2 Long Atoms -

Long atom values are used instead of short atoms if the atom is used in a plex base (4.2.2.3.)

The specifier for a long atom value has:

type: T LATOM

value: Pointer to long atom data block

The long atom data block contains the following fields:

htype: H_LATOM

hlink: Not used (0)

la_value: An integer representing a unique atom identification. Note that since SETL has an infinite integer range, the question of overflow does not arise.

la_form: Pointer to the form table entry of the plex base

la_nlmaps: Number of local maps based on this long atom. For details, see the section on Plex Bases (4.3.2.5.)

la_nwords: Length of data block, including header

The length of the long atom data block header is given by the system constant HL_LATOM.

2.6 Boolean Values

In SETL, TRUE and FALSE are distinguished constant atom values, and are stored in the same manner as any other short atom value. The values for TRUE and FALSE are 0 and MAXSI, respectively. However, when the SETL TYPE operator is applied to booleans, the result is the string 'BOOLEAN'. Furthermore, the I/O routines associate #T and #F with TRUE and FALSE, respectively.

2.7 Undefined Value: Omega

The undefined value is called omega, noted as OM. and treated specially. Its specifier has a TYPE and VALUE which correspond to some proper defined value (of the appropriate type in the case of an object with a REPR). The IS_OM bit of this specifier is set to indicate that the value is undefined.

From the SETL language point of view, there is only one omega value. The use of multiple representations of omega within the library is useful in the case of based map and set formats, where omega values can retain typing information. In addition, the fact that omega values appear to have a proper value of the correct type means that code which omits the IS_OM check always produces results which, though they may not be correct, do not result in failure of system integrity.

3.0 UNTYPED PRIMITIVE DATA

In addition to the typed data structures described in the previous section, there exists untyped data which do not carry a type code. Unlike any other data in the system, such values cannot be identified by the bit pattern of their specifier. To maintain the integrity of the environment so that the garbage collector can operate correctly, the manner in which untyped data can appear is restricted as described in this section.

3.1 Untyped Integers

An untyped integer fits into all or part of a SETL word. It typically corresponds to a signed integer value of the hardware integer size.

3.2 Omega Untyped Integers

The omega untyped integer value is represented by a unique machine dependent bit patterns which meets the requirement that it is never produced as the result of any integer operation on defined values where the result is properly defined and in range. Typical choices are negative zero (1's complement) or the largest negative number (2's complement).

3.3 Untyped Reals

An untyped real fits into all or part of a SETL word. It typically corresponds to a signed real value of the hardware real size.

The format of an untyped real is the same as the format of a real data word (see section 2.3).

3.4 Omega Untyped Real

The undefined untyped real value (untyped real omega) is represented by a unique machine dependent bit pattern which meets the requirement that it is never produced as the result of any real operation on defined values where the result is properly defined. Where an operation on real values produces an improper result (e.g. 0.0/0.0), the omega value must be generated.

3.5 Skip Words

These specifiers do not represent SETL values, but are used to mark untyped primitive data for the garbage collector.

The marking phase of the garbage collector iterates over the symbol table and the (SETL) stack tracing the heap pointers of long data items. A skip word is inserted before each block of untyped data to assure that the garbage collector does not attempt to interpret the bit pattern in the position where typed data items store their type code. Whenever the garbage collector finds a skip word, it ignores or skips the next n specifiers, where n is the VALUE of the skip word.

type:	T SKIP
value:	Number of words to be skipped by the garbage collector. This number includes the skip word.

4.0 NON-PRIMITIVE DATA

This section contains details on the format of non-primitive data items, that is data items which contain other data items as components. Mathematically, these data items correspond to ordered vectors or sets, and unordered sets. Ordered sets are called tuples. The mathematical notion of a relation is captured by the SETL multi-valued map, the notion of a function by the SETL single-valued map. Note that SETL uses the mathematical definition of a relation as a set of pairs for its definition of a map.

Every non-primitive data item consists of a specifier containing a heap pointer to a heap data block with the following standard fields:

hform:	Form table pointer
nelt:	Cardinality of the composite object
hash:	Hash code
is neltok:	Flags valid NELT field
is hashok:	Flags valid HASH field

4.1 Tuple Formats

In addition to the standard fields for non-primitive objects, all tuple data blocks have two standard fields:

maxindx:	Index of the last component allocated
is range:	Used only in map iterators (see section 6.3)

MAXINDX implies the number of words allocated for the tuple data block not counting the words for the tuple header data block. This may exceed the value implied by the NELT field to allow for growth space, in which case the extra words contain the specifier for the appropriate omega value. If the NELT value is correctly set (as indicated by the IS NELTOK setting), then the value corresponding to the index value NELT must not contain omega (so that the NELT value corresponds to the cardinality of a tuple).

4.1.1 Standard Tuple -

A standard format tuple is represented by a specifier which has:

type:	T TUPLE
value:	Pointer to tuple data block

The tuple data block has the standard fields, with

htype:	H_TUPLE

Following the header is a zero-origin vector of data values. These are ordinary value specifiers giving the values of the NELT successive elements of the tuple, or the omega value for indices for which the tuple value is not defined. Note that the zero entry always contains the proper omega value and cannot be modified (since tuples in the SETL language are one-origin). The extra zero index value speeds indexed references.

4.1.2 Special Tuples -

Special tuples (as opposed to standard tuples) are represented by a specifier which has:

type:	T STUPLE
value:	Pointer to appropriate tuple data block

There are three subtypes of special tuples, distinguished by the HTYPE values of the corresponding data blocks.

4.1.2.1 Packed Tuples -

The packed tuple data block consists of a standard format header which has (in addition to the standard fields):

htype:	H_PTUPLE
ptbits:	Number of bits per value
ptvals:	Number of values per word
ptkey:	Specifier used to interpret packed values as SETL values

The values are stored from right to left in each word, fitting as many values as possible in each word. Thus the setting of PTVALS can be computed from the PTBITS setting and the word size. It is stored to save the time for this calculation. If there are unused bits, they are set to zero.

4.1.2.1.1 Packed Values -

The values stored in packed tuples and maps are small integers known as pack indices. These indices are mapped into SETL specifiers by means of a key. The key for a packed tuple is contained in the PTKEY field; the key for a local packed map is kept in its LS KEY field, to be described in the proper sections.

There are two types of keys depending on the REPR of the tuple or map:

packed tuple(integer i..j)

> In this case, I and J must be integer constants, and we assume that the difference J-I is reasonably small. In this case the values I..J are stored as indices 1..J-I+1, and the index zero signifies the undefined value omega. The key has the following format:
>
> type: T_INT
>
> value: I-1

packed tuple(elmt B)

> Here B must be the name of a constant base. The pack indices then correspond to the EBINDX values for the elements of B. In order to retrieve a value from this type of tuple, we provide a standard tuple(elmt B) whose i'th component is the specifier for the base element with EBINDX i. This tuple is internally generated by the compiler (namely the COD phase). Again, a pack index of zero is used to represent the undefined value omega, and the zeroth component of the tuple(elmt B) contains the proper omega value. The key has the following format:
>
> type: T_TUPLE
>
> value: Pointer to standard tuple(elmt B)

4.1.2.2 Real Tuples -

A real tuple contains untyped real values (see section 3.). A real tuple data block has:

htype: H_RTUPLE

A real tuple data block is identical in format to a standard tuple except that the values stored are untyped reals (or the omega untyped value) rather than standard typed values.

4.1.2.3 Integer Tuples -

An integer tuple contains untyped integer values (see section 3.). An integer tuple data block has:

htype: H_ITUPLE

An integer tuple data block is identical in format to a standard tuple except that the values stored are untyped integers (or the omega untyped integer) rather than standard typed values.

4.1.3 Null Tuples -

Unlike the previous system, the null tuple does not have a special value. It is simply a tuple whose data block header word has a zero NELT value with its IS_NELTOK bit set.

4.1.4 Pairs -

A tuple of two non-omega elements is called a pair. It is stored in the same way as an ordinary tuple with the NELT field containing a value of 2 to indicate that two elements are present (i.e. elements with subscripts 1 and 2). Small tuples in general, and pairs in particular, usually do not have growth space allocated. Thus the value in MAXINDX is usually the same as the value in NELT.

4.2 Set And Map Formats

There are several formats for sets and maps corresponding to possible REPR declarations. In this description, the use of the word "set" is reserved for sets other than maps (although the SETL mode for all these objects is set). In addition to the typing of objects as sets or maps by the appearance of relevant REPR declarations, objects are dynamically converted between these formats where appropriate. For example, a set of pairs is converted to map format if it is used as a map, and the inverse conversion occurs if a non-pair element is added to an object stored in map format (since maps can only contain pairs). This dynamic conversion is transparent to the SETL program, since a set of pairs and the corresponding map are semantically equivalent.

Set and map data blocks have several standard fields (in addition to the fields which are common to all non-primitive data):

hashtb:	Pointer to hash table header block
is_based:	Flags local and remote sets and maps
is_map:	True for maps, false for sets
is_mmap:	True for maps which are multi-valued everywhere
is_smap:	True for maps which re single-valued everywhere
is_elset:	True for sets of base elements, and maps from base elements to some range mode

The HASHTB field points to the hash table for the set or map if it is unbased, and to the hash table of the base if it is based. IS_ELSET distinguishes the special cases of a set whose members are elements of some base, or maps whose domain elements are elements of some base. These objects can be treated in an especially efficient way in some situations.

4.2.1 Hash Table Structure -

Sets and maps in various formats are represented by hash tables. The basic format of all hash tables is consistent throughout the data structures, although the manner in which information is stored in the individual element blocks of the hash table depends on the usage.

The hash table pointer (HASHTB) in the set header points to a hash table header data block which contains the following fields:

htype:	H_HT
hlink:	Not used (0)
neb:	Number of element blocks in hash table
lognhedrs:	Log(base 2) of number of hash headers

The number of hash headers is always a power of 2, with a minimum value of 1 (corresponding to a LOGNHEDRS value of 0). The system constant MAX_LOGN defines the maximum hash table size. If the size of the set would require a larger hash table, the expansion request will not be honoured, and gradual performance deterioration will take place as the length of individual clash lists increases.

The NEB field contains the number of element blocks in the hash table, excluding the template block and hash headers. In the case of unbased maps, this value might differ from the NELT value in the set header.

The hash table proper consists of a contiguous sequence of hash table header blocks (HTB), and clash lists of element blocks (EB).

HTB's represent a 'short' form of EB's in the sense that they consist only of the fields needed to iterate over the set, but have no values associated with them. Their size is constant, and is given by the system constant HL_HTB.

There is one EB for each set element, plus one special EB, called the (hash table) template block. The template block immediately follows the hash table header data block in memory. The number of words in each EB varies with the context in which the hash table is used.

The following fields occur in every hash table header block:

htype:	H_HTB
hlink:	Not used (0)
eblink:	Pointer to either a clash list of EB's, or the next HTB
is_ebhedr:	True
is_ebtemp:	False

The following fields occur in every element block:

htype:	Type of hash table EB
hlink:	Not used (0)
eblink:	Pointer to either the next EB in this clash list of EB's, or the next HTB
is_ebhedr:	False
is_ebtemp:	False
ebsize:	Number of words in EB. (This field overlaps HLINK)
ebspec:	Specifier for the set element, or the map domain element

The EBSPEC field is a composite field which includes all fields found in a normal value specifier. It may be the case that EBSPEC corresponds to an entire SETL word, but this is not required.

The EBLINK field is used to chain the EB's of a hash table together in one long linked list, as further described below.

The IS_EBHEDR flag identifies the start of each clash list in the hash table.

The number of SETL words required for these standard fields of an EB (which is the minimum possible size for an EB with no extra fields) is given by the system constant HL_EB.

The template EB occurs immediately following the hash table header word and is a dummy EB with its fields set as follows:

ebspec:	Appropriate omega value
eblink:	Points to the first hash table block (HTB)
is_ebhedr:	True (a special case)
is_ebtemp:	True

The hash table proper consists of a contiguous sequence of element blocks called hash headers. The number of hash headers can be obtained from the LOGNHEDRS field of the hash table header word. The system constant MAX_LOGN gives the maximum number of hash table blocks in any hash table.

The EBLINK field of the template points to the first hash table HTB. Often the template block will immediately precede the hash headers, but this is not required.

Each hash table block is chained (using the EBLINK field) to a list of EB's.

The chain of EB's from one hash table block correspond to those elements which hash to the given header position. The EB's of the hash chain are chained using the EBLINK field of the EB's. The end of each chain links back to the next hash table HTB. The EBLINK field of the last block (either HTB of EB) points back to the template block. Thus the EB's of a set, together with the corresponding HTB's, form one long circular list. The end of the list is detected by testing the IS_EBTEMP flag which is only set for the template block itself.

Every hash table has at least one hash table HTB. In the case of a null set, there will be one hash table HTB.

This format results in the elements of the hash table being chained together in one long list while retaining the possibility of hashed access for a search. Note that the end of a given hash chain is detected by encountering an HTB, marked by the IS_EBHEDR flag.

4.2.2 Set Formats -

All set formats are represented by a specifier which has:

type: T_SET

value: Pointer to set data block

The IS_MAP bit of the referenced data block is always off. The HTYPE of the data block indicates the particular format of set referenced.

4.2.2.1 Unbased Sets -

This section describes the format of unbased sets. Note that this includes sets of elements of a base in the case where neither the keyword REMOTE nor the keyword LOCAL appeared in the REPR. Such objects are stored in standard unbased format with the appropriate specifiers in element-of-base format.

The set header data block has:

htype: H_USET

the HASHTB field of the data block (which contains no fields other than the standard ones) points to a standard format hash table (4.2.1).

The element block has the following fields:

htype: H_EBS

The EBSPEC fields of these element blocks contain the values of successive elements of the set in standard specifier format. The EBSPEC field of the template block contains the omega value typed and formed consistently with the type of element in the set.

The size of each unbased element block is constant, and given by the system constant EBS_NW.

4.2.2.2 Base Sets -

Base sets are represented by a specifier in the standard format. The data block for a base set has:

htype: H_BASE

In addition to the standard fields defined for all set headers, the following three fields are defined for a base set data block:

blink:	Used by the garbage collector to link all bases
rlink:	Used by the garbage collector to link remote objects to their bases
nlmaps:	Number of local (unpacked) maps on the base

The length of this expanded header block is given by the system constant HL_BASE.

The HASHTB field of the base set data block, as well as the HASHTB field of all objects based on it, point directly to the base set hash header data block. The element blocks in a base set hash table contain the following fields:

htype:	H_EBB
eblink:	EB link pointer
is_ebhedr:	Set in hash header EB's
is_ebtemp:	Set for template block only
ebsize:	element block size (overlaps HLINK field)
ebspec:	Value of base set element
ebform:	Pointer to the Form Table Entry of the base
ebindx:	Index value for remote based objects
ebhash:	Element hash code
is_eblive:	Used during base compaction during garbage collection to mark live element base blocks

The EBINDX values range from 1 up and are allocated consecutively as new EB's are added to the base set. These index values are used to reference the required element of remote objects by indexing.

The EBHASH value is always set correctly.

The number of words required for this expanded EB format is given by the system constant HL_EBB.

If there are local objects associated with the base, then additional words may be allocated following the standard fields. These words are used to hold the values of locally based objects as described in the individual sections on locally based objects. Each unique local object is assigned (statically) a given word or bit field in the base set EB to contain the value. Some of these words contain specifiers and must be processed by the garbage collector, others contain bit strings or untyped data which must

be ignored. The entries for standard local maps contain pointers which must be relocated by the garbage collector. All such entries occur immediately after the initial words. The NLMAPS field of the base header data block gives the number of such entries in each EB.

The fields of the template block are set as follows:

htype:	H_EBB
eblink:	Pointer to first hash header
ebspec:	Appropriate omega value
is_ebhedr:	True
is_ebtemp:	True
ebindx:	Next index value to be assigned
ebhash:	Zero (the hash value for omega)

Note that if the index of the template block is used to access a remote object, the index will always be out of range and a omega value will be obtained as required. Remaining words in the template block (corresponding to local object values) are set in a manner appropriate to the particular local object type as described in the individual sections on local maps. Note that the index value 0 is never used. The corresponding zeroth entry in a remote object S is the template for the remote object and is set in the same manner as the hash table template block field in the base for a corresponding local object.

4.2.2.3 Plex Bases -

A special case of base sets are plex bases. A plex base has no hash table, and consists of long atom values only (see section 2.5.2 for details on long atom values). This means that no hashing operation can be performed on such an object. All objects based on it must be based locally on it. All references to elements of an object based on a plex base must be in 'element of base' format, to be described in section 5. As with the format of a standard element block, the local maps represented by standard specifiers (i.e. those which must be processed by the garbage collector) must come first. The field LA_NLMAPS gives the number of such specifiers in each long atom value.

4.2.2.4 Remote Sets -

A remote set data block contains a standard form set header with the following fields:

htype:	H_RSET
rs_maxi:	Maximum index value

This header is immediately followed by a bit string which represents the value of the remote set. The length of this bit string (as given by RS_MAXI) must be at least as great as the maximum index for any value in the base set which is contained in the remote set (plus one since indices start at zero). The k'th bit indicates the membership in the remote set of the base element whose index is k. The bit is set iff the element is in the remote set. The first bit in the string (corresponding to index value 0) is never used, and is always set to 0.

The bits are arranged from right to left in successive words. The last (partially filled) word contains unused high order bits. The system constant RS_BPW gives the number of bits stored in each word.

4.2.2.5 Local Sets -

A local set data block contains a standard form set header with the following fields:

htype:	H_LSET
ls_word:	Offset to word in EB containing value
ls_bit:	Position of membership bit in EB word

LS_BIT is a bit position from the low order end of the word (least significant bit numbered 1). The single bit in the indicated position of the base EB indicates whether the base element is in the local set or not. It is set iff the element is in the local set. The corresponding bit in the template block of the base set hash table is never set.

4.2.2.6 Constant Sets -

Constant sets are stored in exactly the same format as normal sets except that they are never modified (and cannot be converted to map format). For each constant set, an index vector is built for use by packed index values (see section 4.2.2.1.1.). The zeroth element of this vector contains the omega value. Successive elements contain the values of the set elements in set element format. There is no link between this vector and the

hash table since none is ever needed. If the constant set is used as a base, then its hash table contains index values as usual. These index values match the index values used in the index vector. This means that packed values immediately yield the base index without reference to the index vector.

4.2.3 Map Formats -

All maps are represented by a specifier which has:

type: T_MAP

value: Pointer to map data block

The map data block HTYPE determines the particular map format.

4.2.3.1 Map Image Representation -

Although maps are simply sets of pairs in SETL semantics, the internal storage form is substantially different. In particular, in the case of a multi-valued map, the set of pairs with a common head is grouped together. The image of a map for a particular domain value is thus either a single value or a set of values. The latter case must be distinguished from a single value which happens to be a set. This distinction is the function of the IS_MULTI bit which appears in all specifiers (but is only meaningful in this context). If the IS_MULTI bit is off in the specifier representing the image value, then the map is single valued for the corresponding domain value, and the specifier value is the range value.

If the IS_MULTI bit is on in the specifier representing the image value, then the specifier represents the value of a set whose members are the range values corresponding the domain value. If the map is single valued, then two representations are possible. Either the single range value with IS_MULTI off or a singleton set with IS_MULTI on.

The IS_SMAP bit of the map header block is set on if all image values have IS_MULTI off. This also implies that the map is single valued.

The IS_MMAP bit of the map header block is set on if all image values have IS_MULTI on. This does not imply that the map is multi-valued everywhere since some or all of the range sets may be singletons.

If both IS_SMAP and IS_MMAP are off, then either representation could be used at single valued points. However, the library standardises in this case to set IS_MULTI only in the case of multi-valued points.

4.2.3.2 Unbased Maps -

The unbased map data block contains only standard fields and has:

htype: H_UMAP

The hashtb field points to a hash table which contains the map values. The element blocks in this hash table have:

htype: H_EBM

ebimag: Map image value

There is one EB for each unique domain element. Its EBSPEC field contains the specifier for the domain value. The EBIMAG field contains the image value with the IS_MULTI bit indicating the format as described in 4.3.3.1.

4.2.3.3 Remote Maps -

A remote map data block consists of a standard format set header which has:

htype: H_RMAP

The HASHTB field of this header points to the hash table header data block for the corresponding base set.

The header is immediately followed by a standard format tuple (T_TUPLE), complete with a tuple header block. The k'th element of this tuple contains the map image value, with IS_MULTI showing the format as described in 4.7.3.1. The value in MAXINDX (i.e. the tuple length) must be at least as large as the largest index value for which the map is defined on the corresponding element. Note that the tuple which is part of the map data block does not contain a pointer back to the base. Rather, this pointer is associated with the map block itself.

4.2.3.3.1 Remote Packed Maps -

A remote packed map data block has:

htype: H_RPMAP

It is immediately followed by a tuple of values as for an unpacked remote map, except that the tuple is in packed tuple form (4.2.2.1.)

4.2.3.3.2 Remote Real Maps -

A remote real map data block has:

htype: H_RRMAP

It is immediately followed by a tuple of values as for an unpacked remote map, except that the tuple is in real tuple form (4.2.2.2.).

4.2.3.3.3 Remote Integer Maps -

A remote integer map data block has:

htype: H_RIMAP

It is immediately followed by a tuple of values as for an unpacked remote map, except that the tuple is in integer tuple form (4.2.2.3.).

4.2.3.4 Local Maps -

A local map data block has a standard format set header with:

htype: H_LMAP

In addition to the standard fields, the following field is defined:

ls_word: Offset to word in EB containing value

The length of this expanded header is given by the system constant HL_LPMAP.

LS_WORD shows the location of the word in each EB of the base which contains the value of the map. The referenced word contains the map image with IS_MULTI indicating the format as usual (4.3.3.1)

4.2.3.4.1 Local Packed Maps -

A local packed map data block consists of a standard format set header with:

htype: H_LPMAP

In addition to the usual fields, the following fields are defined:

ls_word: Offset to word in EB containing value

ls_bit: Starting bit number for field

ls_bits: Number of bits in field

ls_key: Pointer to value vector

The length of this expanded header block is given by the system constant HL_LPMAP.

The LS_BIT value is the bit number of the low order bit of the field (ls bit numbered 1).

The significance of the referenced bit string value and its relation to the LS_KEY field is the same as for the values in a packed tuple (see section 4.2.2.1.1.).

The corresponding field in the template block of the base set hash table contains all zero bits (the packed representation of the omega value).

4.2.3.4.2 Local Real Maps -

A local real map data block consists of a standard format set header with:

htype: H_LRMAP

Additionally, the following field is defined:

ls_word: Offset to word in EB containing value

The word in the base set EB's contains either the appropriate real value, or the omega untyped real value. The corresponding word in the template block of the base set hash table contains the omega untyped real value.

4.2.3.4.3 Local Integer Maps -

A local integer map data block consists of a standard format set header with:

htype: H_LIMAP

Additionally, the following field is defined:

ls_word: Offset to word in EB containing value

The word in the base set EB's contains either the appropriate integer value, or the omega untyped integer value. The

corresponding word in the template block of the base set hash table contains the omega untyped integer value.

5.0 ELEMENT-OF-SET FORMAT

While the value semantic of SETL does not provide for pointers, they nevertheless exist in a very restricted form, and provide for the efficient access to element blocks in hash tables without the need to perform an actual hashing operation.

Values of this type are represented by a specifier which has the following fields:

type: T_ELMT

value: Pointer to corresponding Base EB

If an object in element-of-set format occurs in a context in which an actual value is required (e.g. addition), then the operation will be performed on the embedded value, which is obtained by extracting the value field (EBSPEC) of the element block pointed to. This 'dereferencing' operation is, of course, recursive, although one can easily convince oneself that the operation only involves an iteration.

Values in element-of-set format can occur under two conditions:

1) wherever the data representation ELMT BASE is used

2) for iterations through maps and sets

They might also be used more generally if data representation declarations were allowed to specify 'element of unbased set' or 'element of domain of unbased map'.

6.0 ITERATOR FORMATS

When iterating through a map or base, a standard value specifier is used to control the iteration. In the cases of sets and maps (but not tuples), it is possible to use this specifier to obtain the current value, as well as to obtain the next value of the iteration. However, the SETL run time system usually maintains two separate iterator values except in certain cases where this optimisation is possible and desirable.

6.1 Set Iterators

Set Iterators are the most general form of iterators in SETL. There forms depend on the object iterated over.

6.1.1 Unbased Set Iterators -

Iterators for unbased sets have the following format:

type: T_ELMT

value: Pointer to corresponding element block

The initialisation for the iterator consists of setting the value field to point to the template block. The iteration is then performed by using the value field to locate the EB containing the pointer to the next EB. If the actual value is required it can be obtained from the referenced element block, and it will often be advantageous to obtain this value once on each iteration and store it in a separate location.

Note that this format corresponds exactly to the 'element of set' format described in the preceding section.

6.1.2 Base Iterators -

The format of a based set iterator is similar to an unbased set iterator. The iterator, however, is in 'element of base' format, and thus represents a value which can be described within the data structure representation sublanguage. Again, the iterator is initialised by pointing to the template block, and iteration proceeds analogous to unbased set iteration.

6.1.3 Unbased Map Iterators -

The format of an unbased map iterator is:

type: T_TUPLE

value: Pointer to pair value

Note that a pair is a tuple consisting of exactly two non-omega components.

The first element of the pair is a pointer to the current domain element block:

type: T_ELMT

value: Pointer to map EB

The second element of the pair is in one of two formats, depending on whether we are at a multi-valued point of the map or not.

If the range of the current domain element is single-valued, i.e. has its IS_MULTI bit set to zero, then the second component of the iterator pair is the range specifier corresponding to the current domain specifier.

If the range of the current domain element is multivalued, the second component of the iterator pair points to another iterator describing the iteration through the range set. In this case, the IS_RANGE bit of the map iterator's tuple header block is set.

6.1.4 Based Map Iterators -

The difference between based and unbased map iterators is identical to the difference between based and unbased set iterators: The first (or domain) component of the pair is in 'element of base' format and thus defined within the data structure representation sublanguage.

6.1.5 Tuple Iterators -

A tuple iterator simply consists of a short integer value giving the index of the current tuple component in the iteration. It is initialised to zero.

6.2 Domain Iterators

If a program specifies directly or indirectly an iteration through the domain of a map or tuple, a special domain iterator format is used which refers to the map or tuple itself (rather than actually creating the domain as a set and iterating through it).

6.2.1 Unbased Map Domain Iterators -

The domain iterator is in standard 'element of map domain' format:

type:	T_ELMT
value:	Pointer to the corresponding element block

6.2.2 Based Map Domain Iterators -

This iterator has the same format as a base set iterator for the corresponding base.

6.2.3 Tuple Domain Iterators -

The domain of a tuple is the set of integers from 1 to the number of elements in the tuple. If an iteration through the domain of a tuple is performed, the corresponding domain iterator is a standard format short integer (2.1.1.).