LITTLE Newsletter # 36

E. Deak
November 1, 1974

Run-Time Consderations for MIDL

This proposal describes a run-time environment, compatible with the current SETL system for MIDL.

To be compatible with the SETLB system, the MIDL run-time environment will utilize the same STACK, HEAP, and garbage collector as the SETLB run-time library (SRTL). All objects in the STACK and HEAP must conform to the word formats described in On Programming, Vol. 1, Item 6.

The target language of the MIDL compiler will be LITTLE. Because the type declarations in MIDL enable the compiler to compute address offsets at compile time, most references to MIDL structures, with the exception of MAPTABLES, can be generated as in-line code. Operations involving SETL objects will become calls to SRTL routines. For several of the more complex MIDL operations, involving storage allocation, conversion between MIDL and SETL objects, and MAPTABLES, new entries must be added to SRTL.

The discussion below details the internal representation of MIDL data objects, and LITTLE code fragments which various MIDL operations generate. We use the following notations, relating to the STACK and HEAP word formats of the system:

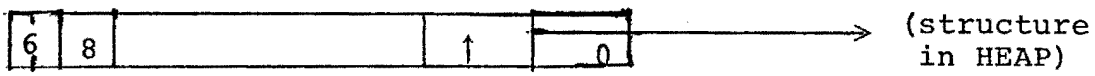
WS	number of bits in a STACK word
POINTER1, PTR1	first pointer field of STACK word
POINTER2, PTR2	second pointer field of a STACK word
POINTER3, PTR3	third pointer field of a STACK word
SWDS	STACK word descriptor size-number of bits needed for the block type field and number of pointers field in a STACK word.
HDRSZ	header area size of a type 0 heap block
PTRSZ	pointer area size of a type 0 block
OFFSTRHDR	number of words in a heap structure header

POINTERS.

A MIDL pointer is an index to the HEAP. A pointer which is not defined has the value UNDEFPTR, which is the system pointer to the undefined word.

Pointers which are components of a structure of more than one component may be packed. However, variables which are declared as pointers are not packed. The variable is allocated one word from the STACK. The pointer is stored in the POINTER1 field of the STACK word allocated to the variable.

A new pointer primitive is to be introduced into SRTL so that pointers may become elements of SETS and TUPLES. MIDL pointers must be converted to SETL pointers before using them as SETL objects. A SETL pointer has a root word of the format:



The SETL pointer will be treated by SRTL as a short blank atom with two pointer fields.

STRUCTURES.

A structure is defined by a TYPE statement. A structure is represented as a block of one or more words in STACK word format, which may or may not contain pointers to objects in the heap. An array of structures is represented as a contiguous block of structures.

The components of a structure are packed. Each STACK word may contain up the three pointers, and a STACK word may accomodate bit strings of length no more than WS - SWDS. A component bit string which is longer than this will become a long structure string, and will be allocated as a separate HEAP block, and a pointer to the HEAP block is stored in the structure. This adds a level of indirection when accessing long structure objects.

Various kinds of structure components may be handled in the following manner:

PTR	A pointer component is mapped onto a STACK word pointer field. Initially UNDEFPTR.
MAP	A MAP component is a pointer to a map table, and is mapped onto a STACK word pointer field. Initially, UNDEFPTR.
SETLOBJ	A machine word is allocated to store a SETL root word. Initially, UNDEFPTR.
BITS(n)	If n .LE. WS-SWDS, then n bits are allocate, initially 0. Otherwise, a pointer field, initiallized to point to allocated HEAP block.
REAL	A REAL number is a long object, and a pointer field is allocated. Initially points to allocated HEAP block.

ENTRY Requires pointer-sized field to contain code address. Each structure component is mapped onto the triple <field, offset, ib>, where-offset-is the word offset from the base of the structure, -field- is the position within the machine word, and -ib- is an indirect bit to flag whether the component is a long string.

Variables associated with structures determine whether the structure is allocated statically or dynamically. If a variable is declared to be a pointer to a structure, the structure is dynamic and is allocated from the HEAP, upon execution of the NEW function. Variables which are themselves structures of more than one component are allocated storage at compile time from the STACK if the structure contains any pointers to the HEAP, otherwise, they become static LITTLE variables.

For example, consider the following definitions:

```
TYPE TP: F1 PTR, F2 SETLOBJ;
TYPE TS: F1 BITS(15), F2 bits(30);
DCL AS TP, AH PTR(TP), BS TS, BH PTR(TS);
```

Both variables AH and BH are pointers to heap structures of type TP and TS respectively. These structures will be allocated dynamically. Variable AS is itself a structure which contains pointers to the HEAP. The structure is allocated storage at compile time (statically) from the STACK. BS is a structure which does not contain pointers to the HEAP, and will be allocated static LITTLE storage.

Structures and arrays of structures in the HEAP which are of size n words, where n is greater than 2, are stored in type-0 HEAP blocks, prefixed by a header of OFFSTRHDR(1) number of words. If n is 1 or 2, the structure is stored in a type-1 or type-2 block respectively. (All arrays of structures of dimension greater than 1 will be stored in type-0 blocks.)

If the HEAP structure itself contains pointer components, the pointer area size PTRSZ is equal to n, and the header area size HDRSZ is equal to 0.

A long structure string of n words will be stored in a type-0 HEAP block, with PTRSZ equal to 0 and HDRSZ equal to n.

For each structure defined in a program, the compiler must compute the internal representation, or template for the structure that is, how many machine words are needed to store the structure and the component-field mapping. The template is computed at the time of the structure type declaration, and is therefore not dependent on the types of variables which are associated with it (eg array, static, dynamic).

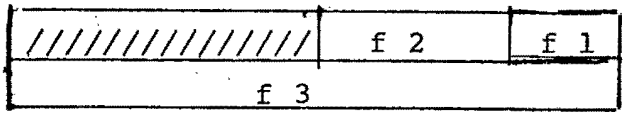
After the template is computed, the compiler will reserve a STACK location to point to a HEAP block which contains the skeleton form of the template(i.e. all PTR fields initiallized to UNDEFPTR and all bit fields initiallized to 0.)

Both statically and dynamically allocated structures will be initiallized to copies of their associated templates. The template for a pointer variable is simply UNDEFPTR. The skeleton template is also used in connection with MAP TABLES, which are described in the next section.

We give a simple example of code that might be produced by a TYPE definition:

```
TYPE TN:  F1 PTR, F2 PTR, F3 SETLOBJ:
```

This structure may be mapped onto 2 machine words, as illustrated below:



We generate the following LITTLE code for this declaration:

```
GET(2, TEMP);  $ get a block of 2-words from the HEAP
TPZZZn = UNDEFPTR;  $ TPZZZn is STACK location for template
PTR1 TPZZZn = TEMP;   HEAP(TEMP) = UNDEFPTR2;
HEAP(TEMP + 1) = UNDEFPTR;  $ SETLOBJ
```

if V1 is a variable of type PTR(TN), the statement V1 = NEW(TN) compiles into

```
STACK(K) = COPY(TPZZZn); $K is the STACK index associated with V1.
```

Static variables are initiallized at point of declaration.
If variable V2 is declared by:

```
DCL V2 TN;
```

the code generated will perform:

```
STACK(K) = UNDEFPTR2;
STACK(K + 1) = UNDEFPTR;
```

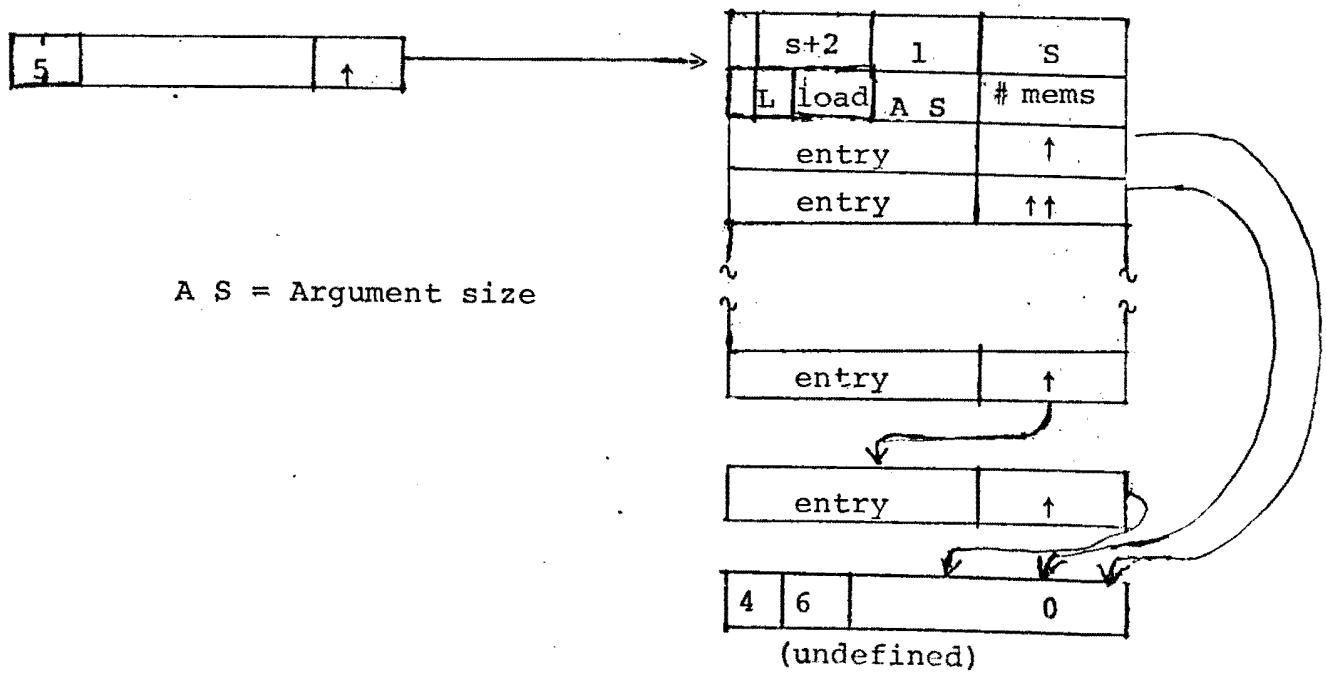
MAP TABLES.

MAP TABLES are stored as hash tables, which grow and shrink in the same way as do SRTL sets. The hash table is accessed indirectly as specified in LITTLE Newsletter 37, via a pointer P in the HEAP, which is updated each time the hashtable is reallocated. All pointers to the hashtable will actually store references to P.

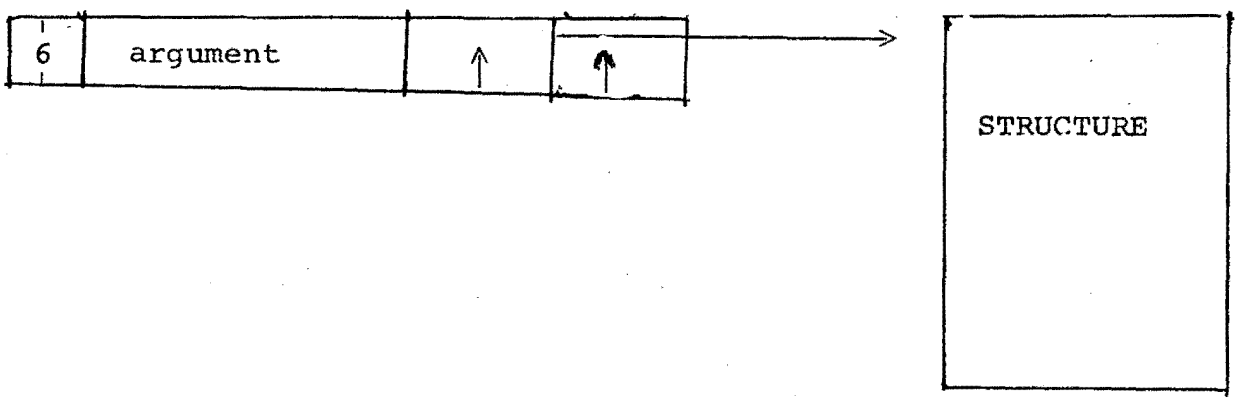
The hash table index is computed from a bit string argument of a fixed, declared number of bits. Each non-empty entry in the hash table stores the argument bit string which indexes that element or if the bit string is too long (greater than WS - 2*PS - SWDS), a pointer to the bit string. A pointer to the structure which is the image of its argument is also stored in a table entry. Entries which hash to the same slot in the table are chained together, as are members of SRTL set. WE reserve a pointer field in each word for this purpose.

If undefined entries are accessed, the skeleton template if returned.

The internal representation for a MAP table is shown below:



Each entry in the table can be illustrated by the following:



VARIABLES AND STACK ALLOCATION MANAGEMENT.

In this section we outline how the compiler allocates storage for variables, and how references for different classes of variables are generated.

First we discuss static bit string variables.

All variables which do not contain pointers to the HEAP become static LITTLE variables. The compiler generates SIZE and DIMS statements corresponding to the MIDL declarations.

For example, suppose we have the declarations:

```
TYPE T:  F1 BITS(30),  F2 BITS(17);
DCL  A  BITS(WS), B T;  DIMS  B(10);
```

The code produced would be:

```
SIZE  A(WS);
SIZE  B(WS);
DIMS  B(10);
```

The reference

```
F2   B
```

would then become:

```
.F.  30  +  1,  17,  B
```

Other variables, that is those which are SETL objects, pointers, maps, and in general store pointers to the HEAP, are allocated STACK locations. (The STACK address allocated is always a negative offset from a base.)

All variables which are either structures or pointers to structures must be initiallized to the skeleton template at the time during execution, when the STACK space is reserved. At the start of execution, STACK space is reserved for global

STACK variables. (The global variable GT points to the top of the global variables area, and global variables are referenced by a negative constant offset from GT)

In the prologue of each routine, STACK space is reserved for local variables and temporaries. Local pointer variables are initiallized at this time. (The variable LT points to the top of the local variable area). When a routine returns, the storage that was allocated for locals and temporaries is released.

Below is exhibited LITTLE code produced from variable references.

Suppose A is a variable which stores a structure with pointers. A reference to A becomes

STACK(T - C_A)

where T is the base for the variable block, and C_A the STACK address assigned to A. If A is dimensioned, and the number of words occupied by each array element is 1, a reference to A(I) becomes:

STACK(T - C_A + I).

An indexed field reference Fl A(I) becomes in LITTLE

.F. C1, C2, STACK(T - C_A + C_3 + I*nwordsA). C1 and C2 specify the field. C3 is the offset of the field from the base of the structure, and -nwordsA- is the number of words needed to store a single structure array element.

Let us now suppose that variable B is a pointer to a structure in the HEAP. Below is code produced by a simple reference B, an indexed reference B(I), and an indexed field reference Fl B(I), where comparable conditions hold as in the example above.

STACK(T - C_B)

HEAP(PTR1 STACK(T - C_B) + OFFSTRHDR + I)

.F. C1, C2, HEAP(PTR1 STACK(T - C_B) + OFFSTRHDR + C_3 + I*nwordsB)

OFFSTRHDR is the offset of the header word of a type-O HEAP block. (if the structure is not longer than 2 words,there is no header.)

If the field F1 in the above example happened to be a word sized long structure string, the resulting code would be:

HEAP(.F. C1, C2, HEAP(PTR1 STACK(T - C_B) + OFFSTRHDR
+ C_3 + I*nwordsB) + OFFLSSHDR)

where OFFLSSHDR is the header offset of a long structure string.

If a variable M is a MAPTABLE, M is referenced through run-time subroutine -SOFMAP- in the case of sinister assignments and -OFMAP- in the case of retrieval. For the expression M(B) we generate

```
ARG1 = M;
ARG2 = TPZZZm;  $ pointer to template of result
CALL OFMAP(B);  $ pointer to result is returned in ARG2
```

For the assignment M(B) = S;
we generate

```
ARG1 = M;
ARG2 = UNDEFPTR;
TEMP = COPY(TPZZZm);  $ obtain copy of template
PTR1 ARG2 = TEMP;
$ Here initialize HEAP(TEMP) to the value of S
CALL SOFMAP(B);
```

PARAMETER PASSING, ENTRY OBJECTS, RECURSION.

The parameter passing mechanism in MIDL is to be the same as in LITTLE. LITTLE passes the address of an actual parameter if the parameter is a simple variable. Otherwise, the address of a temporary is passed.

To get this effect for parameters which are STACK objects, the calling routine will obtain the STACK indices of the actual parameters. These indices will be stored on the top of the STACK. A reference to the ith formal parameter then has the form:

```
STACK(STACK(TL - LL - nfp + i))
```

where LL is the number of words reserved at routine entry and nfp is the number of formal parameters.

In order to implement the ENTRY object feature, the scheme used by the SETL translator system may be adapted. At the beginning of execution, entry constants will be initialized to contain the entry point of each routine. This is easily done by emitting the sequence

```
CALL ENTCON(TEMP);
CALL SUBNAM;
SUBNAM = TEMP;
```

for each routine. -ENTCON- is an assembly routine which will return the address of SUBNAM, obtained from the subsequent CALL statement.

If R is a variable declared to be of type ENTRY, and storing SUBNAM the statement CALL R; will compile into

```
CALL KALLENT(R);
```

-KALLENT- is another assembly routine which will perform a return jump to SUBNAM.

In order to implement recursion, several additional items must be stacked by the prologue of a recursive routine. These include the following:

1) The return address must be stacked. This can be done in the following manner. In the prologue of the recursive routine, we generate a call to a compass routine RASAVE, which obtains the return address and stacks it at a given STACK address.

2) The variables which are static and local to a recursive routine must be stacked. Since these variables do not normally store quantities in STACK word format, we supply an additional stack RSTACK, which will not be scanned by the garbage collector as it will never contain any pointers to the HEAP. References to a local variable A in a recursive routine will become

RSTACK(RT - C_A)

where RT is the top of the RSTACK and C_A the relative address assigned to A.

3) The actual parameters of a recursive routine must be stacked. Upon routine entry, the values of the actual STACK parameters are retrieved and stored in space reserved for parameters. The values of STATIC parameters in recursive routines will be retrieved and stored on RSTACK.