Section II

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

PROJECT MAC

Reply to: Project MAC
545 Technology Square
Cambridge, Mass. 02139

Telephone: (617) 864-6900 x5851

February 16, 1965

DISTRIBUTION LIST:

V. Vyssotsky
J. Couleur
A. Evans
B. Galler
R. Fano
H. Krenn
E. Glaser
F. Corbató
R. Graham
M. Daggett and M Wagner
R. Daley and Otis Wright
S. Dunten and M. Child
[illegible]. Schroeder and L. Pouzin
P. Crisman and D. Oppert
J. Poduska and J. Saltzer
C. Garman and R. Stotz
M. Bailey
D. Edwards

February 16, 1965

To Attached Distribution:

The enclosed Section II of the MAC version of the Design Notebook is an extensive rewrite and simplification of the November 31, 1964 version of segment conventions. As most know from preliminary discussion, the spirit of the mechanism is essentially the same as before but there have been extensive changes and simplifications. Besides the hardware improvements to the augmentor, the biggest changes are conceptual in that the interfaces and notation for the ordinary programmer have been cleaned up. In particular, the "own data" segments and "headers" have been removed and hidden from view, and the call, save and return macro sequences have been shortened and reworked so that undesired features can be stripped away incrementally in special, high-efficiency situations. No generality has been given up in these changes.

Complete coding examples are given in another section so that the interface to an ordinary programmer can now be evaluated.

Section II.

A Proposal for GE 636 Segment Conventions

Introduction

by F. J. Corbató

The purpose of the present memo is to develop a set of suitable conventions, standards and techniques for the use of segments on the GE 636 computer. The current groups interested in standards are: Bell Labs, GE Phoenix, Michigan, Carnegie Tech. and M.I.T. The present proposal is an attempt to be a consensus of the interested members of the Bell, GE and M.I.T. groups.

Background

No effort will be made here to review the segment hardware of the 636, since it is described in early form in MAC memo M-182 by E. L. Glaser and in near final form in the version IV memo of February 3, 1965. (The frozen form of February 10, 1965 is assumed.) The detailed philosophy of segmentation will not be repeated here since it is given in MAC Technical Report TR-11 by J. B. Dennis. However, a brief summary of the advantages of segmentation and paging will be offered in review.

The major reasons for segments are as follows:

1. The user with segments is able to program in a doubly infinite memory system. Thus any single segment can dynamically grow (or shrink) effectively without limit. (A quarter million words maximum with up to a quarter million segments are possible on the 636).
2. The user can operate his program through phases of segment configuration without prior planning of the storage allocation need or the management of the segments.
3. The largest amount of code which must be bound together as a solid block is a single segment. Since binding pieces of code together (i.e. "loading" in today's BSS parlance) is a process which is similar to assembly or compiling, the advantage is immense of being able to prepare arbitrarily large programs out of a series of limited-overhead segment bindings. (c.f. the overhead of Fortran II subprograms vs. Fortran I programs.)

4. Program segments appear to be the <u>only</u> reasonable way to achieve the use of common (i.e. shared among several users) procedures and data bases. Segmentation allows this important goal both elegantly and conveniently.

Pages, such as were first on the Atlas Computer, are a separate idea from segments and have further advantages:

1. The use of a paged core memory allows a very flexible technique of dynamic storage management without the overhead of moving programs back-and-forth in the memory. The importance of this reduced overhead is especially high in heavy-traffic situations such as occur in responsive time-shared systems.

2. The mechanism of paging when properly implemented as in the 636 allows the operation of arbitrarily incompletely paged segments so that by only retaining active pages more effective use can be made of high-speed memory.

<u>Major Features of the Present Proposal</u>

In the present proposal it was felt important to meet the following requirements:

1. Any segment should only have to know of another segment name symbolically. Intersegment binding should occur as needed dynamcially during program execution. Intersegment binding should be automatic (i.e. not explicitly programmed by the user) and the mechanism should operate at high-efficiency after the first binding occurs.

2. Similarly, any segment should be able to reference symbolically a location within another segment. This reference should bind dynamically and automatically; after binding occurs the first time, program execution should be at full-speed.

3. The mechanism should be such that it is straightforward to have <u>all</u> procedures be pure procedures (i.e. capable of being shared by several users).

4. Similarly it should be straightforward to write recursive procedures (i.e. subroutines capable of calling upon themselves either directly or indirectly through a circular chain of calls).

5. The general conventions should be such that the call, save and return sequences used to link one independently compiled procedure to another should not depend on whether or not the two procedures are in the same segment.

<u>Segment Conventions</u>

The output of any translator should by definition be a <u>subprogram</u> which consists of two regions, a <u>pure</u> <u>procedure</u> <u>region</u> and a <u>linkage</u> region. When one or more subprograms are <u>bound</u> together, by a program in a loaded [illegible] called a <u>binder</u>, a <u>segment</u> and an associated <u>linkage</u> <u>section</u> are created. When this is the case, as will be seen later, one or more users can share the operation of a single copy of a pure procedure segment.

As a general rule all segments should have a pair of names, the first a proper name and the second a class name, e.g. ALPHA PUREPRØC, or BETA DATA, etc.

The classes are:

1. Pure procedure
2. Impure procedure — polluted ?
3. Read-write data
4. Read-only data
5. Write-only data

All symbolic names should begin on an integral word boundary and be of variable length fromat where the first 8-bit character is a character count. (Initial implementation may only handle strings of a definite length, for example, of 15.)

Segments are stored in a user's file directory as one type of file and the supervisor can always gain access to segments by appropriately searching the user's directories. Since directories can be tree-structured and can contain indirection links to other files (or to other directories), there is a great searching and linking flexibility possible. Whenever a pure procedure is requested by a user, the supervisor need only have <u>one</u> working copy of the procedure for all users;

however each user will have a private copy of the corresponding linkage section of the pure procedure.

To understand the proposed mechanism better let us consider the limitation of the execution of a process (i.e. thread etc.) by the supervisor program. It is assumed that the user of the system has indicated symbolically to the supervisor a particular segment and internal location at which to start the process. Whenever the supervisor starts a process, it creates several special auxiliary tables for the process. These tables which are all basically hidden from the programmer, are each in the form of segments:

1. Descriptor segment
2. Stack segment
3. Linkage segment
4. Linkage section boundary segment

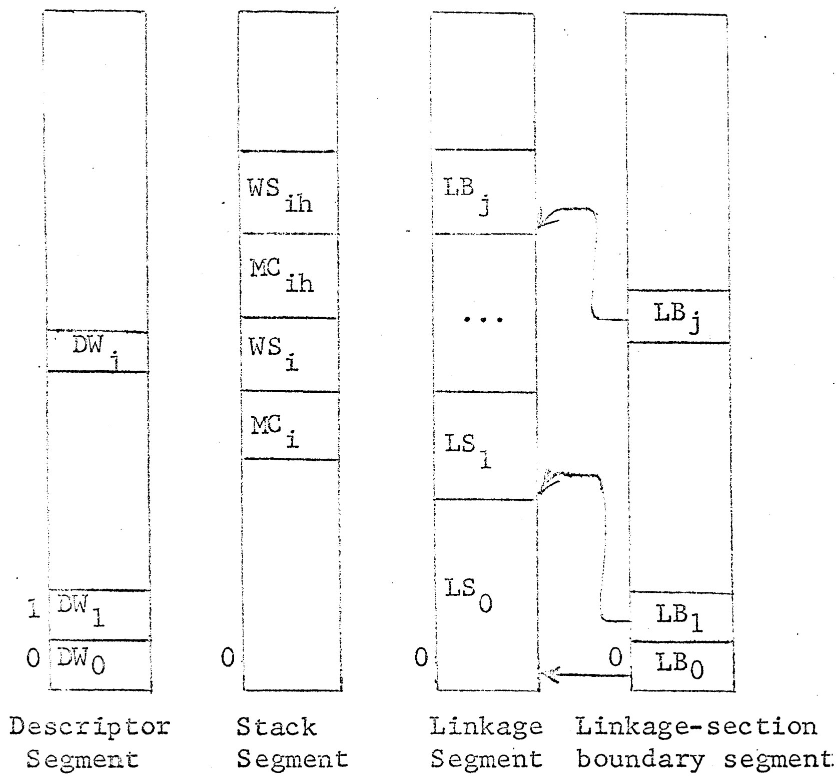
Figure 1 shows these segments. A brief description of each follows:

The descriptor segment contains a sequence of descriptor words for all the segments which have been associated with the process. The descriptor word contains the address of the user's page table for the segment and the descriptor bits which control access to the segment. For reasons which will appear later, the zeroth entry, by convention, should be the descriptor of the "linkage boundary segment".

The stack segment is, as the name suggests, a push-down mechanism or the "scratch pad" or working storage region for every subprogram called. The use of the stack will be explained later.

The linkage segment is built up out of the linkage sections of each of the segments involved in the process. As a process proceeds and the number of segments involved increases, it is expected that the descriptor segment and the linkage segment will grow in length. Automatic page-turning based on activity will prevent the mechanism from becoming unwieldy.

The linkage boundary segment is merely an auxiliary directory to assist in the location of the internal linkage section of any particular segment. Precisely, the $i^{\underline{th}}$ entry is a relative pointer to the beginning of the $i^{\underline{th}}$ linkage section corresponding to the $i^{\underline{th}}$ descriptor in the descriptor segment.



DW_j = Descriptor of jth segment.

MC_i = Machine conditions of ith subprogram call.

WS_i = Work space of ith subprogram call.

LS_j = Linkage section of jth segment.

LB_j = Linkage boundary of jth segment.

FIGURE 1

Notation

To describe the technique of symbolic referencing between segments, it is necessary to use a notation. (This notation, of course, will probably be improved in any assembler which is produced.) To give the address field of an instruction word which has bit 29 on, the notation of (base tag) $\uparrow$ (displacement) will be used. For example, LDA 3 $\uparrow$ 25 or symbolically LDQ sp $\uparrow$ x.

Literals will be designated by an "=" sign.

To describe addresses which are not defined at translation time or at binding time, a notation of brackets is introduced. Thus writing LDA Z$\uparrow$[x] signifies that when the program operates, the A register is to be loaded from the location X in the segment which has its descriptor pointer loaded in base Z. The mechanism of translation, as will be seen, only produces a relative address to an appropriate point within the corresponding linkage section of the program being translated; however as the program operates, the effect will be as indicated.

It is also convenient to refer to the descriptor index of a segment. To refer to a segment descriptor pointer by segment name, one writes <beta> for example. The notation LDA <beta> $\uparrow$ [x] -5, 7 will be used to indicate a similar mechanism. why not [beta] ?

The Linkage Mechanism for Inter-Segment References

Returning to the description of intersegment linking, when a process is initiated, the supervisor "calls" the process starting location with particular base register conventions set. As will be seen subsequent calls made by a procedure within the process use the same conventions. The base register assignment conventions are given symbolically as follows where a suffix b or p designates an external or internal base, respectively:

base	description
sb	pointer to the stack segment descriptor; probably unalterable except by a supervisor call.
sp	pointer to current procedure stack origin
lb	pointer to the linkage segment descriptor
lp	pointer to linkage section of the current procedure
ab	pointer to descriptor of segment containing the argument list
ap	pointer to argument list location

The remaining 2 bases are arbitrarily available for the programmer to use and are labeled bp and bb. All internal bases are assigned to the external base of the same first letter. Bases not explicitly unalterable are alterable.

It should be emphasized that the above base settings are set for convenience upon entry to a procedure. During execution of any procedure, the ap, ab, bp, bb bases are available for any purpose since all machine conditions, including bases, are preserved in the save macro and reestablished in the return macro. With considerable care, 3 more bases can be used for special purposes by saving lp and lb in the stack and then saving sp at sp↑1.

The Call Macro

The call macro is:

```
LDAQ      (argument list pointer)
STAQ      sp↑T + 20
STCD      sp↑18
TRA       <seg>↑[loc]
```

The constant T is the amount of temporary storage required by the procedure. If this amount varies during execution a slightly more elaborate call is required. The argument list pointer can be in the procedure segment or any other arbitrary segment. The location <seg>↑[loc] is automatically established during program execution by a cross-referencing mechanism which is described later. It should be noted that the above call is reasonably economical of space and that error returns, if any, are to be treated as ordinary arguments. If there is no argument list, of course, the call is only 2 instructions. The argument list consists of 2-word ITS-modifier addresses which point to the argument values.

Stack and Linkage Segment Organization

As is indicated in figure 1, the stack is used to store machine conditions. All stack usage by procedures is by increments which are 0 modulo 8. In addition, the assembler will assign all temporary storage within the stack (presumably with a smooth enough programming notation that the user is unbothered by the mechanism). Temporary storage for the program starts at sp↑22 and on up. The amount of temporary storage required by a procedure, T, is by convention always kept as a constant in the first word of the linkage section at lp↑0. Similarly lp↑1 contains the constant -T.

?

[illegible]

Subroutine call argument lists should normally be either in the stack, or the procedure itself (if constant), or in an arbitrary segment; the stack is preferred. Also preferred is the use of argument addresses pointing to values rather than direct argument values: such practices allow sophisticated techniques where values are automatically computed by procedure when needed by means of the "execute pair" modifier mechanism.

The Save and Return Macros

The save and return macros are next presented followed by explanation:

```
            ADBsp    lp↑ 0      set new stack origin
            STB      sp↑ 0      save bases
            STRS     sp↑ 8      save registers
            STCD     sp↑ 16     set new lp
SAVE       {LDX0     sp↑ 16     set new lp
            LDBlp    lb↑ 0,*0   set new lp
            STBsp    sp↑ 0      record info to give effective stack length
            LDCF    (ap→ab)     pair base ap to ab
            EAPap    sp↑ 20,*   establish argument list pointer in ap
```

```
                LDRS      sp$8        restore registers
                LDB       sp$0        restore bases
SAVE            ADBsp     lp$1        reset stack origin
                STBsp     sb$0        record info to give effective stack length
                RTCD      sp$18       return
```

The above macros are for the completely general case of nested calls, recursion, pure procedure, and either inter-segment or intra-segment transfers. It should be obvious that even though the call, save, and return macros are fairly efficient in this general form, they can be further streamlined if there are special conditions such as "no argument list" or "no further calls within the procedure".

The word pair at lb$0 is a pair of constants with an ITS-modifier set by the supervisor when it creates the linkage segment. The pair contains:

```
ARG   0,ITS
ARG   0
```

Similarly the word-pairs at location sp$18 and sp$20 in the stack have the ITS-modifier double-word format.

Within the argument list itself, ITS-modifier word-pairs which point to argument values, matrix origins, etc. can be used for easy, general transmittal. However, storage-saving options are possible at execution-time expense such as using the left half word as a descriptor pointer and the right half word as a location.

In general when passing parameters, it is desirable to pass locations rather than values. The main reason is that this allows mechanisms (such as we will use to mechanize segment cross-referencing) which are triggered by the unavailability of data and which automatically trap to generative procedures.

It is necessary to digress a moment to discuss descriptor segment management. The descriptor segment is initiated by the supervisor whenever it established a process. An obvious algorithm is to assign segment descriptors to successive index values as each new segment reference occurs.

If a process wishes to release for reuse a segment descriptor index value, it should be able to do so by an explicit call to the supervisor; otherwise, the table must grow with the concomitant gradually increasing page overhead.

Within a called procedure, one can manipulate, save or destroy the contents of bases ap, ab, bp, bb. Other bases can be used provided one is careful to restore them before any calls. In particular base sp can be used when no stack references are made between calls and bases lp and lb may be used if no intersegment references are made. To make a reference to an input argument from within a procedure, one can for example use

```
        LDA     ap↑3,5
```

to pick up the 3,5 argument. To refer to a temporary location one can use

```
        LDA     sp↑22+2,5
```

which by appropriate modification of the assembler would be more convenient as

```
        LDA     sp↑X,5
```

Remarks on the Call Linkage

There are special problems whenever a procedure has a requirement for temporaries which is unbounded at translation time. In this case, the subroutine must increment and decrement by an amount equal to 0 modulo 8 lp↑0 and lp↑1, respectively; This should be done by a call to the supervisor to avoid the user having to be able to write in the linkage segment. In addition the call macro used must be special since the value of T is no longer allowed in the pure procedure. In any case, it should be clear that the above complication can usually be avoided and that for non-recursive procedures it is not even necessary to use the stack for more than the basic safestore information, by merely, placing all parameters and temporaries in an arbitrary data segment.

Various refinements deserve brief attention. The save macro can either be repeated for multiple entries or a subroutinized version can be prepared. If one is confident that no registers need be preserved across a call, then two more instructions are saved and other special cases can be worked out. It is hoped that most of the time the general machinery will be used.

It should be observed that the use of the stack mechanism should serve as a major diagnostic tool since whenever a program is stopped, a "traceback" program can work its way back to the base of the stack by using the contents of sp$0.

Branching of a Process into Multiple Processes

Branching must be in invoked by a supervisor call since new descriptor segments, stacks, etc. must be started for each daughter process. The passage of arguments by 2-word pairs represents no problem except that the descriptor index is relative to the parent descriptor segment. Therefore each daughter process must start with a copy of the parent descriptor segment. In addition there is the question of interlocking the reading and/or writing of parent argument values with respect to daughter processes. It should be clear that some additional conventions are required to clarify these issues but that nothing inherently prevents processes branching and rejoining.

Reference Mechanisms

It is now possible to trace through the various addressing mechanisms,

a) Whenever a user wishes to use a constant or make a transfer in the pure procedure section, the assembler should assemble the referencing instruction with bit 29 off (i.e. with ordinary addressing machinery but with paging).

b) Whenever a reference is to the stack of the procedure, the user can write the address in the form of sp$x where sp and x have definite values at translation time. This form of referencing is, of course, limited to a 16K word section in the stack segment unless one explicitly programs the use of an index register, e.g.

```
        LDX2            Y
        LDA             sp↑ 0,2
```

where location Y contains the pointer to location x in the stack.

c) Finally we take up a more general case which illustrates most of the mechanics. Whenever a user wishes to refer to a variable in another arbitrary segment with the descriptor pointer in, say, base 3 he writes in his program, for example

```
        LDA             3↑ [x]-5,7
```

The method used to establish this linkage dynamically at execution time, is to make the reference an indirect one off a location in the procedure's linkage section. In particular, the assembler creates for the original instruction

```
        LDA             lp↑ Z,*
```

where Z is a location in the procedure's linkage section roughly of the form:

```
        Z               ARG     NAM,F          3↑ -5,7
        Z+1             ARG     3$-5,7
                        ...
        NAM             BCI     1,X
```

During execution, the attempt to reference location Z through its address field with an F modifier creates a fault trap to the supervisor. The supervisor can then determine which segment descriptor is loaded into base 3. Let us call this segment "alpha data". Then in the corresponding linkage section of "alpha data" there is a sorted list of all those symbolic names and values which were designated as external at the segment creation time of "alpha data". (The precise form of the linkage section is given later.) When the supervisor has established the value of x as, say, 129, then it rewrites the word at

34124,7

location Z as: ARG 3$124,7. Thereafter, whenever the same reference is made, it proceeds at the normal speed except for the extra indirect cycle.

d) In a very tight loop, or in a case of data segments larger than 16K, the user may wish to use internal bases and thus 1) avoid the indirection cycle within the loop and 2) have general addressing. This is done in a similar manner as above by programming an internal base. In the example it is assumed that base 2 is assigned as an internal base pointing at the appropriate external base.

```
     EAB2          [X],*     (outside loop)
     LDA           2$-5,7    (inside loop)
                   2↑-5,7
```

or alternatively

```
     EAB2          [X-5]     (outside loop)
     LDA           2$0,7     (inside loop)
                   2↑0,7
```

really, I thought this form meant something else.

[X]-5 maybe?

The mechanism of indirection is identical in form as in the previous case, except that the words corresponding to location Z+1 are: ARG -5 and ARG 0 for the two alternative examples given.

e) In the procedure's linkage section, there are similar supervisor trapping addresses for the case of cross-references to other segments. Thus when a user attempts to load an external base with

```
     LDB2          <alpha>
```

the assembled instruction is again to an indirect word at Z where the contents of Z are with a specially coded op code OPCD:

```
     Z    OPCD     (pointer to segment name "alpha"),F
```

The segment name consists of a BCI string in the linkage section. The supervisor is able to rewrite the contents of Z after looking up the corresponding descriptor index of the "alpha" segment. Subsequent references during program execution are at full speed.

Remarks on Referencing

In general the above scheme consists of every program knowing only symbolic information pairs (e.g. symbol x in segment alpha) or invariant indicial pairs (e.g. relative location 1536 in the segment which has its descriptor located in 107 relative of the process descriptor segment). Under no circumstances is a program able to obtain an absolute address of information in memory; thus it is guaranteed that all programs can be invariant to physical storage remapping. Symbolic pairs which are known at translation time are linked automatically to indicial pairs without explicit programming by the user; however there is also a need for a special supervisor call to establish linking in the cases of generated names or names acquired from typewriter input during execution.

In order to endure flexibility arising with "saved" processes it is important that the linking mechanism also be reversable. General program reversability cannot be done but is valuable in special cases (e.g. as in using the latest copy of the library "sine" routine); rather stringent usage requirements are needed for this substitution technique to work correctly. One mechanism for unsnapping links would be to replace each linkage segment with a fresh copy.

Remarks on Segment Creation by the Binder

In the process of binding one or more subprograms into a segment, there arises the need to equate segment names, decide on segment names for the various groupings of subprograms envisioned, etc. This should probably be an interactive program with the user responding at a console. For routine, repetitive binding action, the binder should allow a mechanism for a declaration file to be created by the user.

For convenience there should be a special entry to the supervisor for the case of a program during execution creating empty segments of either given or temporary names. Non-repeating names can be generated from the information; processor factory serial number, date, month, year, and time in units comparable to memory cycles.