Feb. 25, 1965

Section III.

A Proposal for a Minimal Assembler, GAP, for the GE 636

by R.M. Graham

It will be assumed that the reader is familiar with Section II of the Design Notebook (A Proposal for GE 636 Segment Conventions) and Appendix C (Memo of J. Couleur on GE 636 hardware).

Table of Contents

0. Goals

This proposal represents what is considered to be the minimum modifications and additions to the GEM Assembler in order to produce a satisfactory minimal assembler for the GE 636. It should be understood that this assembler, GAP, is to be considered as having only temporary life until an assembler designed specifically for the GE 636 is written. Nevertheless there are certain properties which it must have in order to be a useful tool.

a) It must conform to certain basic time-sharing system (TSS) standards

i) Since all console I/O is in terms of a single, full character set, using 8-bit codes, GAP must be aware of only 8-bit characters.

ii) All input and output must be to and from files and all character input and output must be TSS standard printable files.

iii) GAP itself must be a pure procedure segment (or group of segments) and must be a subprogram callable using the standard GAP call macro.

b) The output of a GAP assembly must be a segment and its associated linkage section.

c) It should be extremely easy to write pure procedures in GAP and it should be difficult to write impure procedures.

d) Input should be free field rather than the existing fixed fields of cards.

e) GAP must be able to process TSS standard names, i.e. variable length format (although imitially this may be limited to a maximum of 15 characters).

These are critical goals, for if GAP does not have these abilities it will be virtually impossible to assemble the time-sharing system itself.

1. <u>Assembler Input-Output</u>

Input to GAP will be a file with class name GAP. For the purpose of this write-up assume that the name of this file is ALPHA GAP. Output will consist of four files: 1) the text file with name ALPHA TYPE where TYPE is supplied by the SEGMENT pseudo-operation which must be the first line of any assembly, 2) the ~~listing~~ Linkage file with name ALPHA LINKAGE, 3) the listing file with name ALPHA LISTING, and 4) the debug-symbol table file with name ALPHA DEBUG.

The three files ALPHA GAP, ALPHA LISTING, and ALPHA DEBUG will conform to the TSS standards for printable files, i.e., a file composed of variable length lines which end with a carriage return and the first 9-bit field of each line giving the number of 8-bit characters in the line (which are right justified in 9-bit fields packed four per word). With regard to ALPHA GAP, each line (composed of less than 512 characters) corresponds to a card in GEM. The fields in the line are separated by the "tab" character, i.e., the GAP instruction format is:

location field	"tab"	op-code	"tab"	address field	"tab"	comment field

The file ALPHA DEBUG contains, as a bare minimum, the symbol table accumulated during assembly. Other information which the debug routine finds useful is to be added later.

The file ALPHA TYPE contains the text (instructions and/or constants) resulting from this assembly. All assemblies will be relocatable, however, initially (until the binder is defined and coded) the relocation information will be discarded and output will be a contiguous block beginning at relative location 0. This can be accomplished by appending to GEM a modified version of the standard loader which would load the text just assembled and separate out the information to form the file ALPHA LINKAGE. The loaded text is then the desired output for file ALPHA TYPE. The file ALPHA LINKAGE has the following structure:

Relative location	Contents
0	ARG T
1	ARG -T
2	ARG k
3	ZERØ PS1,VALUE1
4	ZERØ PS2,VALUE2
...	...
k+2	ZERØ PSk,VALUEk
k+3	SEG P1,F
	ARG C
...	...
PS1	STRING L1,SYMBOL1
PS2	STRING L2,SYMBOL2
...	...
P1	STRING Lj,segment name
...	...

T = the amount of temporary storage needed in the stack by ALPHA, which must be equal to 0 modulo 8. Why not be simple minded and say "a multiple of 8"

k = the number of symbols defined in ALPHA which can be referred to from other segments. Location 3 thru k+2 contain the definition of these symbols. PSi is a pointer to the i^{th} symbol and VALUEi is the symbol's value.

The linkage information which starts at k+3 contains pointers to the segment names and symbols (which are not defined in this assembly) to which ALPHA refers. The symbols and names start at PS1 and use the new STRING pseudo-operation (see section 3).

2. Operation Codes, Bases and Modifiers

The following correspondence of bases will be established and the symbols ap,ab,...,sb will be predefined in the assembler to have the indicated values.

base	symbol	use
0	ap	argument pointer (internal base paired with ab)
1	ab	argument base (external base)
2	bp	free base
3	bb	free base
4	lp	linkage pointer (internal base paired with lb)
5	lb	linkage base (external base)
6	sp	stack pointer (internal base paired with sb)
7	sb	stack base (external base)

The following new operation codes will be added to the assembler:

LBRap	SBRap	EAPap	EABap
LBRab	SBRab	EAPab	EABab
SBRbp	SBRbp	EAPbp	EABbp
LBRbb	SBRbb	EAPbb	EABbb
LBRlp	SBRlp	EAPlp	EABlp
LBRlb	SBRlb	EAPlb	EABlb
LBRsp	SBRsp	EAPsp	EABsp
LBRsb	SBRsb	EAPsb	EABsb
ADBap	STPap	TSBap	LXL0
ADBab	STPab	TSBab	LSL1
ADBbp	STPbp	TSBbp	LXL2
ADBbb	STPbb	TSBbb	LXL3
ADBlp	STPlp	TSBlp	LXL4
ADBlb	STPlb	TSBlb	LXL5
ADBsp	STPsp	TSBsp	LXL6
ADBsb	STPsb	TSBsb	LXL7
SXL0	LDBR	LDCF	
SXL1	SDBR	CLAM	
SXL2	LDB	STAM	
SXL3	STB	STAMY	
SXL4	SCU		
SXL5	RCU		
SXL6	STCD		
SXL7	RTD		

The following psedo-operation codes are to be disabled:

ERLK	SYMREF	LBL	ABS
FUL	TCD	PUNCH	BLOCK

The following new pseudo-operation codes will be added:

SEGMENT	TEMP	STRING
INSERT	TEMPD	EXIT

The pseudo-operation STRING is defined in section 3 and EXIT in section 5. Since a pure procedure may not have any temporary storage within itself TEMP and TEMPD are used for declaring temporary storage in the stack. Single words are defined by TEMP and double word pairs (the first word to be at an even location) are defined by TEMPD. A vector may be defined by enclosing its length in parenthesis after the symbol which is to be defined as its base. For example,

```
TEMP        a,b,c(3)
TEMPD       x,y(2),z
```

would give the following stack assignemnt,

relative location	symbolic location	
0	a	
1	b	
2	c	
3	c + 1	c vector
4	c + 2	
5		
6	x	
7	x + 1	x
8	y	
9	y + 1	y vector
10	y + 2	
11	y + 3	
12	z	
13	z + 1	z

The SEGMENT pseudo-operation must be the first line in the assembly. It defines the type of segment being assembled. For example:

```
SEGMENT        ppsegment
```

would define this segment to be a pure procedure segment. The output text file would then be named ALPHA PPSEGMENT.

The INSERT pseudo-operation is used to insert into this assembly (in place of the INSERT) the contents of another file. For example:

```
INSERT         BETA
```

will insert the contents of BETA GAP and the assembly will continue as if the contents of BETA GAP had actually been written in place of the line INSERT BETA.

The assembler should recognize the following new modifiers:

ITS	EP	CI6
ITB	SC6	F35

the SC6, CI6, and F35 modifiers are to have meaning identical to the SC, CI, and F modifiers in GEM. The SC and CI modifiers are to be redefined to refer to 9-bit fields rather than 6-bit fields (which will be referred to by SC6 and CI6). The F modifier is to be redefined to mean the 636 IT-type fault rather than the 635 IT-type fault (which is now F35).

3. Character Size

When are upper + lower case letters equivalent?

The assembler should consistantly deal with 8-bit characters. That is, all input and output of characters will be 8-bit characters. the BCI and H-type literals must use 8-bit characters. The codes for the 8-bit characters will be those defined as TSS standard character codes. The 8-bit characters are packed four per word and right justified in 9-bit subfields.

The STRING pseudo-operation is identical to the BCI pseudo-operation except that the first 9-bits of the first word of the constant contain a count of the number of characters in the constant with the constant itself starting in the second 9-bit field.

4. Addresses

The following new forms of address are now legal:

(i) BASE ↑ EXP
(ii) BASE ↑ [SYMB]±EXP
(iii) <SEG>↑ EXP
(iv) <SEG>↑ [SYMB]±EXP

where BASE is a symbol, defined at assembly time, with value 0,1,...,or 7 (i.e., specifying one of the base registers), EXP is any legal GEM expression defined at assembly, SEG is the name of some segment, and SYMB is a symbol not defined at assembly time but defined in some other segment. The "± EXP" is optional in (ii) and (iv), and EXP is optional in (i) and (iii). In case (ii) SYMB must be defined in the segment whose descriptor pointer is in the base register BASE while in case (iv) SYMB must be defined in the segment SEG. In all cases the portion of the address following the ↑, when completely evaluated, defines a relative address in the segment referred to by the symbol proceeding the ↑. All address forms may be modified by any valid modifier.

In case (i) all parts of the address are bound at assembly and the resultant 18-bit address is defined as:

bits
0-2 = the value of BASE truncated to 3 bits.
3-17 = the value of EXP truncated to 15-bits.

In cases (ii), (iii), (iv) at least one part of the address is unbound until execution, hence linkage information must be assembled into the linkage section.

Assembly of the various forms results in an assembled address of;

lp ↑ LSP,*

where LSP is the relative location in the linkage section of the following linkage information.

original address	original linkage section entry	linkage section entry after linking
B ↑[SY]-EXP,M	LSP ADD B,F	LSP ADD B, ITB
	ARG	ARG v(SY)-EXP,M
	ARG SSP	ARG SSP
	ARG -EXP,M	ARG -EXP,M
	...	...
	SSP STRING n,SY	SSP STRING n,SY
<SEG> ↑-EXP,M	LSP SEG SP,F	LSP SEG v(SY),ITS
	ARG -EXP,M	ARG -EXP,M
	...	...
	SP STRING k,SEG	SP STRING k,SEG
<SEG> ↑[SY] -EXP,M	LSP SGAD SP,F	LSP SGAD v(SEG),ITS
	ARG	ARG v(SY)-EXP,M
	ARG SSP	ARG SSP
	ARG -EXP,M	ARG -EXP,M
	...	...
	SP STRING k,SEG	SP STRING k,SEG
	SSP STRING n,SY	SSP STRING n,SY

B (a base)and EXP(any valid GEM expression) must be defined at assembly time. M is any valid modifier. SEG is the name of a segment (k characters long) and SY a symbol defined in some other segment (n characters long). v(SEG) is the descriptor pointer for the segment SEG and v(SY) is the value of the symbol SY. The pseudo-operations ADD, SEG, and SGAD are defined to have different values so that the linker will be able to tell which of the three cases exists when it establishes the link.

It is also possible to use a literal of one of these addresses. The literal is written as the address with "=" as a prefix. The assembled address is the same and the linkage section entry is also the same except that it is preceeded by the two words

```
PTR  *+2,F
ARG
```

For example, = ⟨SEG⟩↑-EXP,M produces;

	original linkage section entry			linkage section entry after linking	
LSP	PTR	*+2,F	LSP	PTR	*+2
	ARG			ARG	
	SEG	SP,F		SEG	v(SY),ITS
	ARG	-EXP,M		ARG	-EXP,M
	...			...	
SP	STRING	k,SEG	SP	STRING	k,SEG

The following new symbolic addresses are used in the CALL and EXIT macros and are explained in section 5:

@SYMB   ?n   %A   %Q

5. CALL,SAVE,RETURN, and EXIT Macros

The CALL,SAVE, and RETURN macros will be redefined and a new macro, EXIT, added. There are three variants of the CALL macro:

a) CALL ENTRY

In this call there are no arguments. The macro expands to;

```
STCD    sp↑18
TRA     ENTRY
```

b) CALL ENTRY (@ARGLIST)

In this call the argument list is located at ARGLIST and must have been explicitly constructed by the user. The following code is generated as part of the prologue;

```
EAPbp   ARGLIST
STPbp   sp↑AL
```

and the macro expands to;

```
LDAQ    sp↑AL
STAQ    sp↑T+20
STCD    sp↑18
TRA     ENTRY
```

c) CALL ENTRY (ARG1,ARG2.,,,.ARGn) n≥1

In this call the arguments are explicitly written and GAP will generate the argument list. The following code is generated as part of the prologue;

```
EAPbp    sp↑AL+2
STPbp    sp↑AL
EAPbp    ARG1
STPbp    sp↑AL+2
EAPbp    ARG2
STPbp    sp↑AL+4
...
EAPbp    ARGn
STPbp    sp↑AL+2*n
```

and the macro expands to;

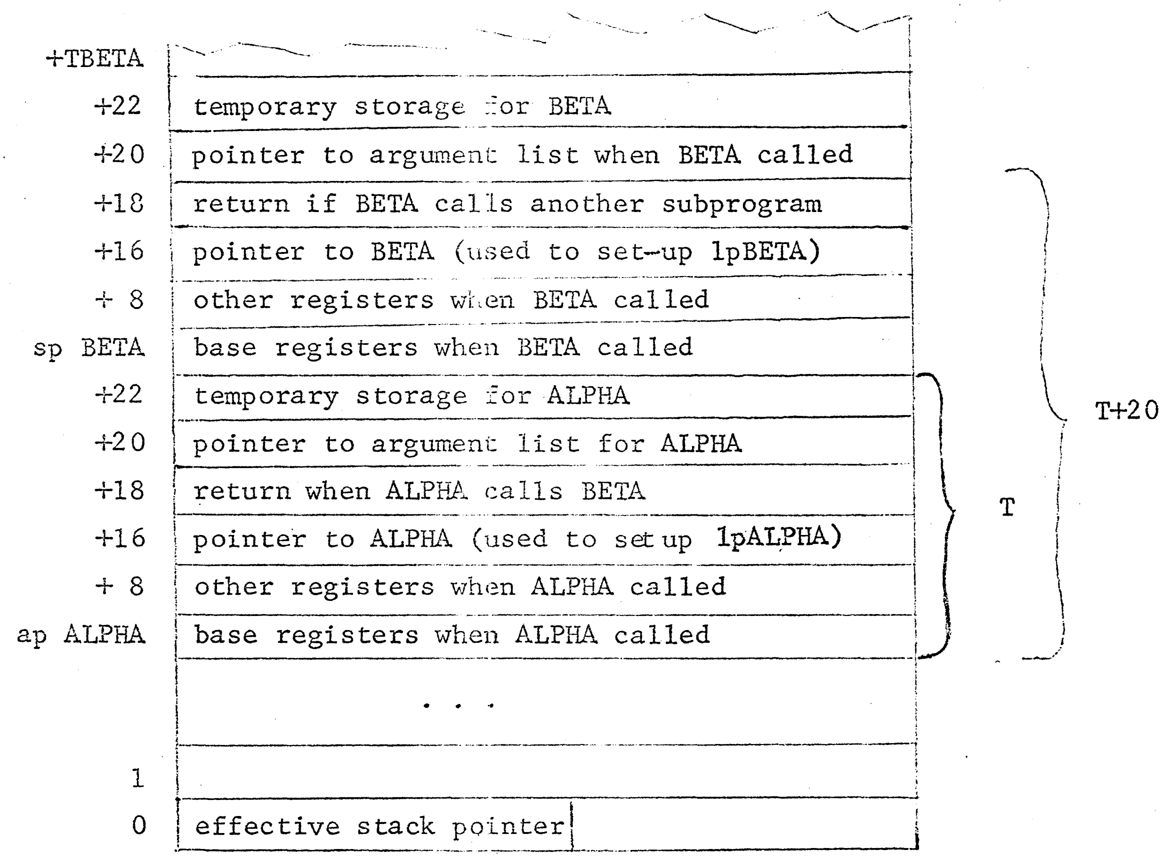
```
LDAQ     sp↑AL
STAQ     sp↑T+20
STCD     sp↑18
TRA      ENTRY
```

In all cases ENTRY is the entry point of the subroutine being called which is usually of the form <SEG>↑[LOC], although the call will work even if ENTRY is a location within this segment. The return is stored in relative location 18 of the stack before transferring to the subroutine. Remembering that the segment being assembled is ALPHA suppose we are calling <BETA>↑0, then the use of the stack is as shown in figure 1. In cases b) and c) a pointer (an ITS pair) to the argument list (called the argument list pointer) is passed to BETA by storing it in BETA's part of the stack, i.e., in location 20 relative to spBETA. The pair of instructions,

```
EAPbp    <SEG>↑[LØC]-EXP
STPbp    sp↑AL
```

will produce in sp↑AL and sp↑AL+1 the pair of words;

0–17	18–29	30–35
v(SEG)		ITS
v(LØC)-EXP		



stack segment

spALPHA = contents of base sp when executing ALPHA

spBETA = contents of base sp when executing BETA

lpALPHA = contents of base lp when executing ALPHA

lpBETA = contents of base lp when executing BETA

T = amount of stack storage needed by ALPHA

TBETA = amount of stack storage needed by BETA

Figure 1

The argument list itself is a list of pointers which point to the actual arguments. In case b) the user must construct his own list and in case c) the argument list is constructed in the prologue, the code being generated by the CALL macro (see section 7 on implementation). The prologue (which may be viewed as an addition to the SAVE macro) is executed once upon entry to the subprogram. ARGLIST in case b) may be any valid GAP address (e.g., sp ↑ k, <SG> ↑ [x],...). In case c) the ARGi may be any valid GAP address (e.g. sp ↑ k, = 5,...) or a symbol of the form ?m which refers to the m^{th} argument of ALPHA. In this case the instruction pair in the prologue to generate the pointer must be;

```
EAPbp     ap ↑ 2*(m-1),*
STPbp     sp ↑ AL+k
```

The SAVE should be the first instruction executed upon entry to a subroutine. The macro expands to;

```
ADBsp     lp ↑ 0           set sp to spBETA
STB       sp ↑ 0           save bases
STRS      sp ↑ 8           save registers
STCD      sp ↑ 16          set lp to lpBETA
LDXO      sp ↑ 16                ...
LDBlp     lb ↑ 0,*0              ...
STBsp     sb ↑ 0           save effective stack pointer
LDCF      (ab<-ap),DU      pair pase ap to ab
EAPap     sp ↑ 20,*        get argument list pointer
```

Referring to figures 1 and 2, the stack pointer sp is incremented by T so that it is now pointing to BETA's area in the stack. The bases and registers are saved and the linkage pointer lp is reset to lpBETA so that it points to BETA's linkage section. The STCD instruction stores the two word pair:

sp + 17	loc(STCD)+2		
sp + 16	β	indicators	ITS
	0 17	18 29	30 35

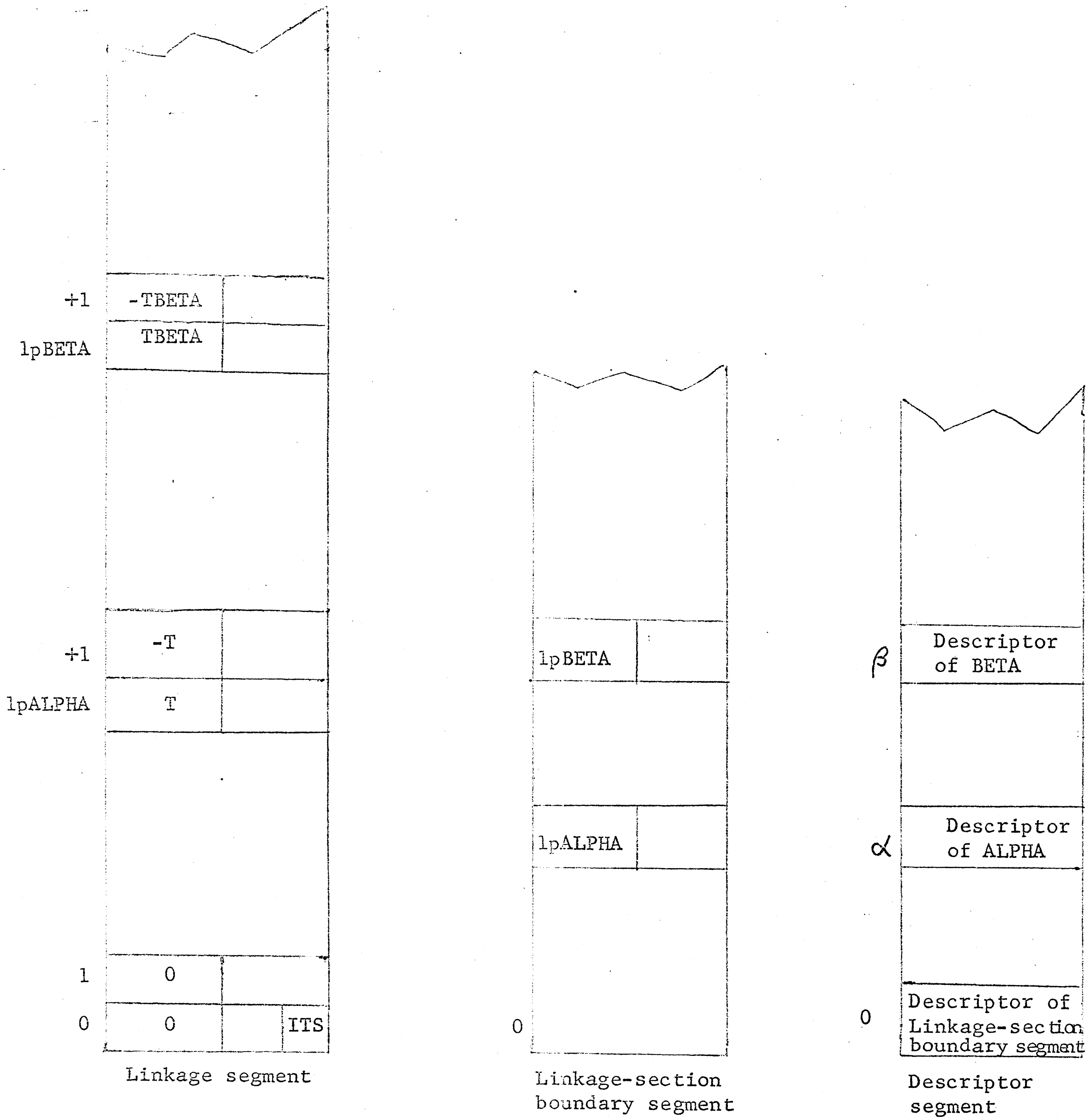

+1
-TBETA
lpBETA
TBETA
+1
-T
lpALPHA
T
1
0
0
0
ITS
Linkage segment
0
lpBETA
lpALPHA
Linkage-section boundary segment
β
Descriptor of BETA
α
Descriptor of ALPHA
0
Descriptor of Linkage-section boundary segment
Descriptor segment

Figure 2

where β is a pointer to the descriptor of BETA and loc(STCD) is the relative location (in BETA) of the STCD instruction. An ITS pair pointing to the linkage-section boundary segment is always located at relative location zero in the linkage segment. The descriptor of the linkage-section boundry segment is always at relative location zero in the descriptor segment. The contents of sp (which now contains spBETA) is saved in relative location zero of the stack. This allows the supervisor to compute an upper bound on the effective stack length when the stack contents need to be saved. That is, $C(sb \uparrow 0)_{0-17}$ = spBETA, sb $\uparrow$ spBETA+16 leads to lpBETA, and $C(lb \uparrow lpBETA)_{0-17}$ = TBETA. The upper bound on the effective stack length is then spBETA + TBETA + 20. The instruction LDCF (ab$\leftarrow$ap),DU sets the control bits of the bases ab and ap such that ab is an external base and ap is an internal base which is paired to ab. The argument list pointer is then loaded into the pair of bases (ab$\leftarrow$ap).

The RETURN macro is used to return to the calling program. The macro expands to;

LDRS	sp $\uparrow$ 8	restore registers
LDB	sp $\uparrow$ 0	restore bases
ADBsp	lp $\uparrow$ 1	reset sp to spALPHA
STBsp	sb $\uparrow$ 0	reset effective pointer
RTCD	sp $\uparrow$ 18	return

This is effectively the inverse of the SAVE macro.

The EXIT macro is used for error returns and alternate exits from the subprogram. Error returns and alternate exits are always given in the argument list just as any other argument. An EXIT macro is necessary to insure that the stack and linkage pointers are correctly reset. The macro,

EXIT	?n

expands to;

LDXO	2*(n-1),DU	move exit pointer to
LDAQ	sp $\uparrow$ 20,*0	fixed location in
STAQ	sb $\uparrow$ 2	stack
LDRS	sp $\uparrow$ 8	restore registers

```
LDB       sp ↑ 0     restore bases
ADBsp     lp ↑ 1     reset sp to spALPHA
STBsp     sp ↑ 0     reset effective stack pointer
LDI       sp ↑ 18    restore indicators
TRA       sb ↑ 2     exit
```

The pointer to the exit location in the calling program is moved from the argument list to a fixed location, namely relative location 2 in the stack segment, and the machine conditions are restored as in the SAVE macro. The indicators have to be explicitly restored.

As can be seen from the macro expansions all registers are restored before returning to the calling program. Hence, if it is desired to return to the calling program with a result in either the A or Q registers (or both) the result must be placed in the proper stack locations (sp ↑ 8 for the A register and sp ↑ 9 for the Q register) so that it will be loaded by the SAVE or EXIT macro. To facilitate this the following symbols will be defined in GAP which will refer to the proper stack location:

symbol	register
%A	A
%Q	Q

6. Coding Examples

a) The first example is a subprogram to calculate N! (the factorial of N) by recursion. It illustrates that the standard CALL, etc., macros use the stack in such a way that it is trivial to write recursive subprograms. The call is to be:

CALL <FLIB>↑[FACT](N,ERREXIT)

The subprogram is located in the segment named FLIB with entry point FACT. The result N! is to be left in the A register or, if $N! \geq 2^{35}$, the subprogram should exit to ERREXIT.

```
         SEGMENT  FLIB
         SYMDEF   FACT
FACT     SAVE
         LDA      ap↑0,*        get N
         TNZ      OVER          if N=0
         LDA      1,DL            then
         STA      %A              N!=1
         RETURN
OVER     STA      N             if N≠0
         SBA      1,DL            N1=N-1
         STA      N1
         CALL     <FLIB>↑[FACT](N1,EX)    find (N-1)!
         LLR      36            switch (N-1)! to Q
         MPY      N             N*(N-1)!
         LLS      36            test for
         TRC      EX              overflow
         STA      %A            return N! in A
         RETURN
EX       EXIT     ?2            overflow, take error exit
         TEMP     N,N1          reserve N,N1 as temps in the stack
         END
```

Following is the complete contents of the segment FLIB, i.e., the previous subprogram with all macros expanded, all segment address symbols, etc., replaced by the proper base register references, and all pseudo-operations dropped.

```
FACT    ADBsp   lp↑0
        STB     sp↑0
        STRS    sp↑8
        STCD    sp↑16
        LDX0    sp↑16              SAVE macro
        LDBlp   lb↑0,*0
        STBsp   sb↑0
        LDCF    (ab←ap),DU
        EAPap   sp↑20,*
        LDCF    (bb←bp),DU
        EAPbp   sp↑24
        STPbp   sp↑22
        EAPbp   sp↑29              prologue
        STPbp   sp↑24
        EAPbp   EX
        STPbp   sp↑26
        LDA     ap↑0,*
        TNZ     OVER
        LDA     1,DL
        STA     sp↑8
        LDRS    sp↑8
        LDB     sp↑0
        ADBsp   lp↑1               RETURN macro
        STBsp   sb↑0
        RTCD    sp↑18
OVER    STA     sp↑28
        SBA     1,DL
        STA     sp↑29
        LDAQ    sp↑22
        STAQ    sp↑52              CALL macro
        STCD    sp↑18
        TRA     lp↑4
        LLR     36
        MPY     sp↑28
        LLS     36
        TRC     EX
        STA     sp↑8
        LDRS    sp↑8
        LDB     sp↑0
        ADBsp   lp↑1               RETURN macro
        STBsp   sb↑0
        RTCD    sp↑18
EX      LDX0    2,DU
        LDAQ    sp↑20,*0
        STAQ    sb↑2
        LDRS    sp↑8
        LDB     sp↑0               EXIT macro
        ADBsp   lp↑1
        STBsp   sp↑0
        LDI     sp↑18
        TRA     sb↑2
```

The linkage section for this segment is:

```
        ARG     32
        ARG     -32
        ARG     1
        ZERO    p1,0
        SGAD    p2,F
        ARG
        ARG     p1
        ARG     0
p1      STRING  4,FACT
p2      STRING  4,FLIB
```

A partial picture of the stack shows its use during the first two calls on FACT:

	k+ 0			save bases for 1st call
	8			save registers for 1st call
	16	d(FLIB) FACT+5	ITS	pointer to loc(STCD)+2
	18	d(FLIB) OVER+7	ITS	return for 2d call
	20			arg list ptr from original call
	22	d(STACK) k+24	ITS	arg list ptr for 2d call
first call	24	d(STACK) k+29	ITS	pointer to N1 } argument list
	26	d(FLIB) EX	ITS	pointer to EX } argument list
	28	N		temp N
	29	N1		temp N1
	30			
	j=k+32			save bases for 2d call
	j+ 8			save registers for 2d call
	16	d(FLIB) FACT+5	ITS	pointer to loc(STCD)+2
	18	d(FLIB) OVER+7	ITS	return for 3d call
	20	d(STACK) k+24	ITS	arg list ptr from 1st call
	22	d(STACK) j+24	ITS	arg list ptr for 3d call
2d call	24	d(STACK) j+29	ITS	pointer to N1 } argument list
	26	d(FLIB) EX	ITS	pointer to EX } argument list
	28	N		temp N
	29	N1		temp N1
	30			
	k+32			save bases for 3d call

b) This example is a subprogram to compute the integral of a function using the trapezoid rule. The call is to be,

CALL ⟨INTEGRATE⟩↑[TRAPZ](A,B,N,F)

The subprogram is to be located in the segment INTEGRATE with entry point TRAPZ. The function to be integrated is F, and it is to be integrated over the interval from A to B in N equal steps.

```
        SEGMENT  INTEGRATE
        SYMDEF   TRAPZ
TRAPZ   SAVE
        FLD      ap↑2,*
        FST      B             save B
        FSB      ap↑0,*        compute
        FDV      ap↑4,*          H=(B-A)/N
        FST      H
        CALL     ap↑6,*(?1)    compute F(A)
        STA      S
        CALL     ap↑6,*(?2)    compute F(B)
        STA      TM
        FLD      TM
        FAD      S
        ADE      =-1B25,DU     (F(A)+F(B))/2
        FST      S
        FLD      ap↑0,*        get A
        TRA      TEST
AGAIN   CALL     ap↑6,*(X)     compute F(X)
        STA      TM
        FLD      TM
        FAD      S
        FST      S
        FLD      X             compute
TEST    FAD      H               X=X+H
        FST      X
        FCMP     B             if X≥B
        TMI      AGAIN           compute
        FLD      S               S*H
        FMP      H               as
        FST      %A              integral
        RETURN
        TEMP     X,H,S,B,TM
        END
```

The linkage section for the segment INTEGRATE is:

```
    ARG      40
    ARG      -40
    ARG      1
    ZERO     p1,0
p1  STRING   5,TRAPZ
```

The prologue is:

```
LDCF    (bb←bp),DU
EAPbp   sp↑24       arg list pointer
STPbp   sp↑22          for first call to F
LDAQ    ap↑0        ?1
STAQ    sp↑24
EAPbp   sp↑28       arg list pointer
STPbp   sp↑26          for 2d call to F
LDAQ    ap↑2        ?2
STAQ    sp↑28
EAPbp   sp↑32       arg list pointer
STPbp   sp↑30          for 3d call to F
EAPbp   sp↑34       X
STPbp   sp↑32
```

A partial picture of the stack used by INTEGRATE is (where the value of sp after the SAVE is k):

```
k+ 0  bases
   8  registers
  16
  18  return for F
  20  pointer to arg list in calling program
  22  arg list pointer for 1st call to F (sb↑k+24)
  24  pointer to A (copied from caller)
  26  arg list pointer for 2d call to F (sb↑k+28)
  28  pointer to B (copied from caller)
  30  arg list pointer for 3d call to F (sb↑k+32)
  32  pointer to X (sb↑k+34)
  34  X
  35  H
  36  S
  37  B
  38  TM
  39 ..not used..
  40  start of stack area used by F
```

The following is a subprogram which calls on INTEGRATE. It's call is:

```
     CALL   <SINAREA>↑[ENTRY](A,B)
```

It calculates and prints the integral of sine(x) over the interval from A to B.

```
        SEGMENT   SINAREA
        SYMDEF    ENTRY
ENTRY   SAVE
        CALL      <INTEGRATE>↑[TRAPZ](?1,?2,=100.,<TRIG>↑[SIN])
        STA        RSLT
        CALL      <IOPK>↑[PRINT](RSLT)
        RETURN
        TEMP      RSLT
        END
```

Its prologue is:

```
     LDCF    (bb←bp),DU
     EAPbp   sp↑24         arg list pointer for
     STPbp   sp↑22            <INTEGRATE>↑[TRAPZ]
     LDAQ    ap↑0          ?1
     STAQ    sp↑24
     LDAQ    ap↑2          ?2
     STAQ    sp↑26
     EAPbp   =100.
     STPbp   sp↑28
     EAPbp   lp↑8,*         <TRIG>↑[SIN]
     STPbp   sp↑30
     EAPbp   sp↑34         arg list pointer for
     STPbp   sp↑32            <IOPK>↑[PRINT]
     EAPbp   RSLT
     STPbp   sp↑34
```

Its linkage section is:

```
        ARG     40
        ARG     -40
        ARG     1
        ZERO    p1,0
        SGAD    p2,F
        ARG
        ARG     p3
        ARG     0
        SGAD    p4,F
        ARG
        ARG     p5
        ARG     0
        SGAD    p6,F
        ARG
        ARG     p7
        ARG     0
p1      STRING  5,ENTRY
p2      STRING  9,INTEGRATE
p3      STRING  5,TRAPZ
p4      STRING  4,TRIG
p5      STRING  3,SIN
p6      STRING  4,IOPK
p7      STRING  5,PRINT
```

c) This example is a matrix multiply subprogram. The call is:

CALL ⟨MTX⟩↑[MPY](N,M,A,B,C)

The product of the two matrices A and B is stored in C. The matrices are NXN and are stored row-wise as submatrices of MXM (M≥N) matrices, i.e., A(i,1) is M words away from A(i-1,1). The following subprogram calls MTX:

```
   SEGMENT MAIN
   SYMDEF  IN
IN SAVE
   CALL  ⟨MTX⟩↑[MPY](⟨DATA⟩↑[DIM],⟨DATA⟩↑[DIM]+1,⟨A⟩↑,⟨A⟩↑5000,⟨C⟩↑)
   RETURN
   END
```

The two matrix dimensions are found in consecutive locations in the segment DATA. The two matrices are both in segment A, one starting at A+0 and the other at A+5000.

The linkage section is:

```
     ARG       40
     ARG       -40
     ARG       1
     ZERO      p1,0
     SGAD      p2,F
     ARG
     ARG       p3
     ARG       0
     SGAD      p2,F
     ARG
     ARG       p3
     ARG       1
     SEG       p4,F
     ARG       0
     SEG       p4,F
     ARG       5000
     SEG       p5,F
     ARG       0
     SGAD      p6,F
     ARG
     ARG       p7
     ARG       0
p1   STRING    2,IN
p2   STRING    4,DATA
p3   STRING    3,DIM
p4   STRING    1,A
p5   STRING    1,C
p6   STRING    3,MTX
p7   STRING    3,MPY
```

The multiply routine calls on a subprogram to compute the dot product of two vectors. This is done to illustrate the manipulation of argument pointers and the construction of an argument list. The call to the dotproduct is:

```
        CALL   ⟨VECT⟩↑[DOTPROD](N,M,ROW,COL)
```

ROW is the base of a row of consecutive elements and COL is the base of a column of elements seperated from each other by M words.

```
        SEGMENT  MTX
        SYMDEF   MPY
MPY     SAVE
        LDAQ     ap↑0          get ptr to N
        STAQ     AL
        LDAQ     ap↑2          get ptr to M
        STAQ     AL+2
        LDAQ     ap↑4          get ptr to A
        STAQ     AL+4
        EAPbp    ap↑8,*        load base with ptr to C
        LDX0     ap↑0,*        get N (row count)
        TRA      TEST
NXTROW  LDAQ     ap↑6          get ptr to B
        STAQ     AL+6
        LDX1     ap↑0,*        get N (col count)
NXTCOL  CALL     ⟨VECT⟩↑[DOTPROD](@AL)   mpy row of A
        STA      bp↑0                      by col of B
        ADBbp    1,DU          C base+1
        LDA      1,DU
        ASA      AL+7          B ptr+1
        SBX1     1,DU          col count-1
        TNZ      NXTCOL
        LDA      ap↑2,*        get M
        ASA      AL+5          A ptr+M
        SBX0     1,DU          row count-1
TEST    TNZ      NXTROW
        RETURN
        TEMPD    AL(4)
        END
```

The linkage section is:

```
      ARG      32
      ARG      -32
      ARG      1
      ZERO     p1,0
      SGAD     p2,F
      ARG
      ARG      p3
      ARG      0
p1    STRING   3,MPY
p2    STRING   4,VECT
p3    STRING   7,DOTPROD
```

And the prologue is:

```
      LDCF     (bb←bp),DU
      EAPbp    sp↑24
      STPbp    sp↑22
```

The dotproduct routine is:

```
         SEGMENT   VECT
         SYMDEF    DOTPROD
DOTPROD  SAVE
         LDCF      (bb←bp),DU
         LDA       ap↑0,*        get N
         STA       N
         LDA       ap↑2,*        get M
         STA       M
         EAPbp     ap↑6,*        col ptr to base
         EAPap     ap↑4,*        row ptr to base
         STZ       %A            zero sum
         LDX0      0,DU          zero col count
         LDX1      0,DU          zero row count (i=0)
NEXT     FLD       bp↑0,0        col(i)*row(i)
         FMP       ap↑0,1
         FAD       %A            accumulate sum
         FST       %A
         ADX0      M             col count+M
         ADX1      1,DU          row count+1 (i+1)
         CMPX1     N             test if done i>N
         TRC       NEXT
         RETURN
         TEMP       N,M
         END
```

March 30, 1965

DISTRIBUTION

E. Glaser, MIT (3)
A. Evans, Jr., Carnegie Tech.
[redacted]
J. Ossanna, Bell Telephone Lab. (3)
R. Reeves, Ohio State University
R. Boennighausen, GE
R. Claussen, GE (2) for E. Jacks at GMTC
G. Cowin, GE for SDC
M. Dustin, GE
I. Epstein, GE
W. Estfan, GE (6)
W. Gutman, GE
W. Heffner, GE
J. Merner, GE
G. Oliver, GE
F. Raunikar, GE (2) for Dr. Laird at Penn State
R. Ruth, GE
H. Sassenfeld, GE (8)
G. Scott (3)
W. Shelly, GE
E. Vance, GE
J. Weil, GE
R. Turner, NASA Lewis Res. Center

March 30, 1965

H. M. Sassenfeld, Manager
Project MAC-BTL

Attached is a brief description of the Bootstrap Assembler prepared by Bill Steiner and Branton Ratcliff. The content is as agreed upon by MAC, BTL and GE people in the meeting at MIT on March 22, 1965. We have started the implementation.

R. C. McGee/tas

R. C. McGee
Project MAC-BTL

RCMcG:tas

636 BOOTSTRAP ASSEMBLER

This document describes an assembler to be run on the GE-635 which will produce GE-636 object code. It will be used as the basic implementation tool for producing 636 software until the 636 assembler is available in late 1965. Its companion, the 636 simulator, will allow 636 programs to run on the 635 before 636 hardware availability.

Although the "Bootstrap Assembler" will require restrictions to be adhered to which will be relaxed in the 636 assembler, it will provide the software developer with the following important capabilities:

1. A method of expressing addresses in the segmented environment of the 636.

2. The ability to use 8 bit ASC II code (with restrictions) in assembly source language and assembly outputs.

3. The ability to express the new instructions and address modification types of the 636.

4. A free field syntax which is equally well suited for use at typewriter terminals or conventional input devices.

5. A method of producing object code in the proper format to become input to the binder.

The bootstrap assembler for the 636 will consist of GEMAP preceded by a preprocessor and followed by a post processor. The function of the preprocessor is to translate 636 code into a form acceptable to GEMAP and from which GEMAP can directly generate 636 code. The function of the post processor is to take the GEMAP output and convert it into 636 format, i.e., to take the binary output and split it into a procedure segment and a linkage section and to convert the listing output to TSS standards.

The preprocessor reads 9 bit characters and normally compresses it into corresponding 6 bit characters except when special data generating pseudo operations are encountered. In addition when 636 type code is encountered, the code is translated so as to look like 635 code (to GEMPA) but behave like 636 code. It should be noted that with the exception of pre-declaration statements each line of code is translated at the time it is encountered and without examining its effect on earlier or subsequent code lines. The consequence is that the programmer should be aware of the translation process when preparing MACROs. To facilitate generating linkage information the preprocessor makes use of multiple location counters. As a result the user would be well advised to avoid multiple location counters. The preprocessor generates instructions with 636 addresses by using the VFD pseudo operation. To avoid making operation code names reserved symbols, a table of octal equivalents and mnemonic codes is stored in the preprocessor. Hence the OPD and OPSYN pseudo operations cannot be used.

1. Assembler Input/Output

The following files are considered:

SOURCE	Input to Bootstrap Assembler
PROC	Text resulting from the assembly
LINKAGE	The linkage file for PROC
LISTING	The listing file for PROC
DEBUG	The symbol table for PROC

The files SOURCE, LISTING and DEBUG will conform to TSS standards for printable files.

The file, SOURCE, will consist of statements of the form

label: √ OPCD √ address, mod ↓ comment c_r

Under certain conditions, any or all of the fields may be null.

label

The label field consists of from 1 to 6 characters if present. The label may be preceded by one or more blanks and must be immediately followed by a colon (:). The label may be explicitly null (a colon preceded by no blanks or some number of blanks).

Symbols may consist of alphanumeric characters plus decimal point.

OPCD

The operation code may consist of from 0 to 6 characters. If null, it must be explicitly null through the presence of a comma delimiter (or a leading open parenthesis in the address field). The OPCD may be preceded by one or more blanks.

The operation code is terminated by a comma, a group of one or more blanks, or a statement terminator.

variable

The variable field, if not null, may consist of one or more subfields. The variable field of a machine instruction may include an address field followed by a modifier field. Either may be null.

Each subfield may be preceded by one or more blanks and must be terminated by a comma, a group of one or more blanks, or a statement terminator.

terminator

The statement terminator may include a comment in which case the first character is @ Characters within the comment may be any of the Bootstrap Assembler restricted character set except @.

The comment is terminated by @, or is implicitly terminated by a CR, semicolon (;), or EOR.

If no comment is present, the statement is terminated by a CR, semicolon (;), or EOR.

address

The address field may be of the following forms:

<SG> ↑ [SY] ±EXPR

<SG> ↑ ±EXPR

BASE ↑ [SY] ±EXPR

BASE ↑ ±EXPR

±EXPR

The first form may be written in the form

SY ±EXPR

if the symbol SY has been predeclared to be in segment SG by its appearance in a SEGREF statement, as in

SEGREF SG, A, B, SY, ---

The segment name appears first and is followed by the sequence of symbols within that segment. Additional SEGREF statements may be used with the same segment name appearing as the first symbol.

2. Operation Codes, Bases and Modifiers

The bootstrap assembler will establish the following correspondence between symbolic bases and GE-636 address bases:

Symbol	Base	
AP	0	argument pointer
AB	1	argument base
BP	2	free base
BB	3	free base
LP	4	linkage pointer
LB	5	linkage base
SP	6	stack pointer
SB	7	stack base

With n representing the symbols listed above, the following new operation codes will be available:

LBRn	Load base register n
SBRn	Store base register n
EAPn	Effective address to pair
EABn	Effective address to base n
ADBn	Add to base n
TSBn	Transfer and set base
STPn	Store pair
LXLx	Load index from lower $0 \leq x \leq 7$
SXLx	Store index in lower $0 \leq x \leq 7$
LDBR	Load descriptor base register
SDBR	Store descriptor base register
STB	Store bases
LDB	Load bases
LDCF	Load control field
SCU	Store control unit
RCU	Restore control unit
STCD	Store control double
RTD	Return control double
CLAM	Clear associative memory
STAM	Store associative memory
STAMZ	Store associative memory zero

The following pseudo operation codes are to be disallowed:

ERLK	TCD	BLOCK
FUL	LBL	REM
SYMREF	PUNCH	HEAD
SYMDEF	ABS	OPSYN
		OPD

*Cards are also disallowed.

The following new pseudo operation codes will be added:

TEMP	SEGMENT
TEMPD	TALLYB
SEGDEF	ACC
SEGREF	ACI
ITS	ITB

The following modifiers will be added:

ITS	Indirect to segment
ITB	Indirect to base
EP	Execute pair
FI	Fault immediate

The pseudo operations TEMP and TEMPD are used to declare temporary storage in the stack.. Thus, the pseudo-operations

```
TEMP    A, B, C(3)
TEMPD   X, Y(2), Z
```

would allow the assembler to effectively generate, for example,

SP ↑ 0	for	A
SP ↑ 2+K	for	C+K
SP ↑ 6	for	Y-2

The pseudo operations ACI and ACC generate ASCII characters in 9 bit data fields. Data strings may be generated either with or without a character count. The statements

```
ACI   6, STRING
ACI   "STRING"
```

will each generate a word containing STRI followed by a word containing NGƀƀ.

The statements

```
ACC    6,STRING
ACC    "STRING"
```

will each generate a word containing 6STR followed by a word containing ING b. The first 9 bits contain a binary value which is the number of meaningful characters. In the statements

ACI   k, $c_1c_2c_3 \ldots c_k$

ACC   k, $c_1c_2c_3 \ldots c_k$k, c

k is the number of meaningful characters.

The SEGMENT pseudo operation must be the first line in the assembly. The output text file will be given the name and segment type specified in the variable field.

The pseudo operation ITS is used to generate an indirect pair (ITS). The statement

```
ITS    EXPR,MOD,SEGP
```

will generate an ITS modifier word in the next available even location. The address field will contain SEGP, the effective base address pointer. The adjacent higher odd word will contain address EXPR and modifier MOD.

Similarly, the pseudo operation ITB is used to generate an ITB pair.

```
ITB    EXPR,MOD,BASE
```

In this case BASE is the base address register which contains the effective pointer.

SEGDEF is used to declare those symbols defined within the segment which may be referenced externally. Thus,

```
SEGDEF    SY1, SY2, ...
```

Each symbol appearing in SEGDEF results in the creation of an entry in the linkage section. SEGREF is used to declare external symbols which will be referenced internally. Thus,

```
SEGREF    SEG1,A,B,C,D
SEGREF    SEG2,X,Y,Z
```

The first subfield contains a segment name. The remaining symbols are symbols the user expects to be in the specified segment. The SEGREF statement is not required for symbols which are always declared in use as being external. Thus, if the external symbol X is always referred to as <SEG2> ⇑ [X] it is not necessary that X also appear in a SEGREF statement. The symbol, X, not bracketed and not mentioned in a SEGREF statement would be considered an internal symbol. If X appears in more than one segment via SEGREF statements, its unbracketed use would cause the assembler to select the one first appearing in the assembly.

3. Characters

All input to the assembler will be in the form of 9-bit characters. However, certain limitations will be placed in the form of a limited character set for symbolic information and remarks. The full ASCII set will be allowed in literals and data defining pseudo ops, but other restrictions are placed on the use of data fields.

The existing 6-bit data capability of GEMAP will be retained. Corresponding capability will be added for 9-bit data. Thus, the following language additions.

1. ACI and ACC pseudo operations.

2. The following VFD subfield has been added:

   $An/c_1c_2...c_k$

   This subfield for ASCII corresponds to the existing H subfield but does not replace it.

3. Literal type A for ASCII has been added and is similar to type H. When count is missing, 4 is assumed. Count must be ≤ 4.

4. A literal type V subfield has been added which corresponds to 2 above.

   ASCII literals and subfields are limited to one word of generated information.

   If either ACI or ACC appears in a macro definition region, it must contain no substitutable arguments (#).

Except in 9-bit A-type literals or data defining pseudo-ops, the following are the only acceptable characters. In data fields (ACI, A-type literals, etc.) the full ASCII is acceptable.

Character	6-bit Code	9-bit Code
0	00	060
1	01	061
2	02	062
3	03	063
4	04	064
5	05	065
6	06	066
7	07	067
8	10	070
9	11	071
[	12	133
#	13	043
@	14	100
:	15	072
>	16	076
b̸	20	040
A	21	101
B	22	102
C	23	103
D	24	104
E	25	105
F	26	106
G	27	107
H	30	110
I	31	111
&	32	046
.	33	056
]	34	135
(	35	050
<	36	074
\	37	134
↑	40	136
J	41	112
K	42	113
L	43	114
M	44	115
N	45	116
O	46	117
P	47	120

Character	6-bit Code	9-bit Code
Q	50	121
R	51	122
-	52	055
$	53	044
*	54	052
)	55	051
;	56	073
'	57	047
+	60	053
/	61	057
S	62	123
T	63	124
U	64	125
V	65	126
W	66	127
X	67	130
Y	70	131
Z	71	132
←	72	137
,	73	054
%	74	045
=	75	075
"	76	042

Actually, CR, TAB and EOR are also meaningful to the assembler outside 9-bit data fields, but cannot be used in 6-bit data fields or remarks.

4. Linkage

Linkage at execution time is required for the following external address type:

1. OP <SEG> ↑ [SYM] *EXPR, M

2. OP BASE ↑ [SYM] *EXPR, M

3. OP <SEG> ↑ EXPR, M

In all cases the generated instruction will be of the form:

OP LP ↑ Z, *

The generated linkage information will be:

ITB/ITS pairs / Auxiliary Information

1\. Z

W		FI
±V(EXPR)	TYPE 1	M

W

SYMP	
SEGP	

2\. Z

W		FI
±V(EXPR)	TYPE 2	M

W

SYMP		
Base		

3\. Z

W		FI
V(EXPR)	TYPE 3	M

W

-	
SEGP	

where SYMP and SEGP are defined by

SYMP ACC "name of symbol"

SEGP ACC "name of segment"

Linkage will transform the "ITB/ITS" pairs to

1\.

V(SEG)		ITS
V(SYM) ±V(EXPR)	TYPE 1	M

2.

<table>
<tr><td>Base</td><td></td><td></td><td>ITB</td></tr>
<tr><td colspan="2">V(SYM) ±V(EXPR)</td><td>TYPE 2</td><td>M</td></tr>
</table>

3.

V(SEG)		ITS
V(EXPR)	TYPE 3	M

Future expansion of the linkage scheme to include features such as linking several "names" at one time, unsnapping links, or object time generation of segment names can be accomplished by augmenting the TYPE codes and utilizing the lower half words of the auxiliary information.

The following external addresses are permitted for macro-call arguments:

(<SEG> ↑ [SYM] ±EXPR, M)

(BASE ↑ [SYM] ±EXPR, M)

(<SEG> ↑ EXPR, M)

(BASE ↑ EXPR, M)

The entire address which includes any modifier must be enclosed within parenthesis. The preprocessor will make the following substitution for the Z-th argument of this type

(Q_i, *)

Q_i is generated outside the macro as

Q_i: ARG original argument with parenthesis stripped

Map of Linkage Section

Relative Location	Contents	Description
0	ARG T	length of stack
1	ARG -T	
2	ARG K	number of internal symbols
3	ZERO PS1, VS1	pointer to name and values of internal symbols referenced externally
4	ZERO PS2, VS2	
...	...	
k+2	ZERO PSK, VSK	
	EVEN	ITx pairs for reference to external symbols and segments
	ARG W1, FI	
	TYPE i_1, V(EXP1), M1	
	ARG W2, FI	
	TYPE i_2, V(EXP2), M2	
	...	
	ARG WN, FI	
	TYPE i_N, V(EXPN), MN	
W1	ARG SYMP1	auxiliary information for ITx pairs
	ARG SEGP1	
W2	ARG SYMP2	
	ARG SEGP2	
...	...	
WN	ARG SYMPN	
	ARG SEGPN	
SYMP1	ACC "SYMBOL 1"	names of external symbols and segments
SEGP1	ACC "SEGMENT 1"	
SYMP2	ACC "SYMBOL 2"	
SEGP2	ACC "SEGMENT 2"	
...	...	
SYMPN	ACC "SEGMENT N"	
SEGPN	ACC "SEGMENT N"	
PS1	ACC "INTERNAL SYMBOL 1"	names of internal symbols referenced externally
PS2	ACC "INTERNAL SYMBOL 2"	
...	...	
PSK	ACC "INTERNAL SYMBOL K"	

BOOTSTRAP ASSEMBLER

REVISION I

The linkage section format has been revised to include sufficient information to unlink segments. The content of the new linkage section is as follows:

LK.IN references to this segment are to be defined by following this pointer

LK.OU unused at present; however, this pointer is intended to describe information concerning external references made by this segment.

ID code identifying the class of information its associated pointer is referencing.

INT.N pointer to a block of symbol pointers and their associated values. The symbols are used as references to symbols in this segment.

K number of words in the block pointed at by INT.N

PS_i pointer to an internal symbol possibly referenced in another segment.

VS_i value associated with internal symbol

Z_i indirect reference generated by a reference to an external symbol or segment. The general form of the indirect reference is:

OP LP ↑ Z_i , *

W_i location of auxiliary information associated with Z_i. For the bootstrap assembler there will be exactly one W_i generated for each Z_i .

FI fault code which causes an immediate interrupt

** vacant address to be filled by the linker during execution

M_i address modifier coded with a reference to an external symbol or segment.

PW_i location of a triplet of words identifying the type of linkage required and containing pointers to external segment and symbol names. Duplicate triplets will be suppressed. A pointer indicating the last occurring W_i referencing the triplet will also be stored in the triplet.

$V(EXPR_i)$ value of the expression coded with a reference to an external segment or symbol.

W_{i-1} location of auxiliary information making the last previous reference to a triplet Pw_i. By using the pointer, W_{Ni}, in the Pw_i triplet and successively following the pointers, W_{i-1}, a trace can be made through every reference to the triplet Pw_i. The last W_{i-1} in this chain is zero.

TYPE code indicating the required linkage for the triplet

$SEGN_i$ segment name pointer

$SYMN_i$ symbol name pointer

W_{Ni} pointer to the last occurring reference to a triplet

"name" literal value of the indicated name. The format is:

$$\beta \; S_1 \; S_2 \ldots S_\beta$$

where β is a binary character count in a 9-bit subfield and the S_i are the 9-bit characters with $\beta \leq 6$.

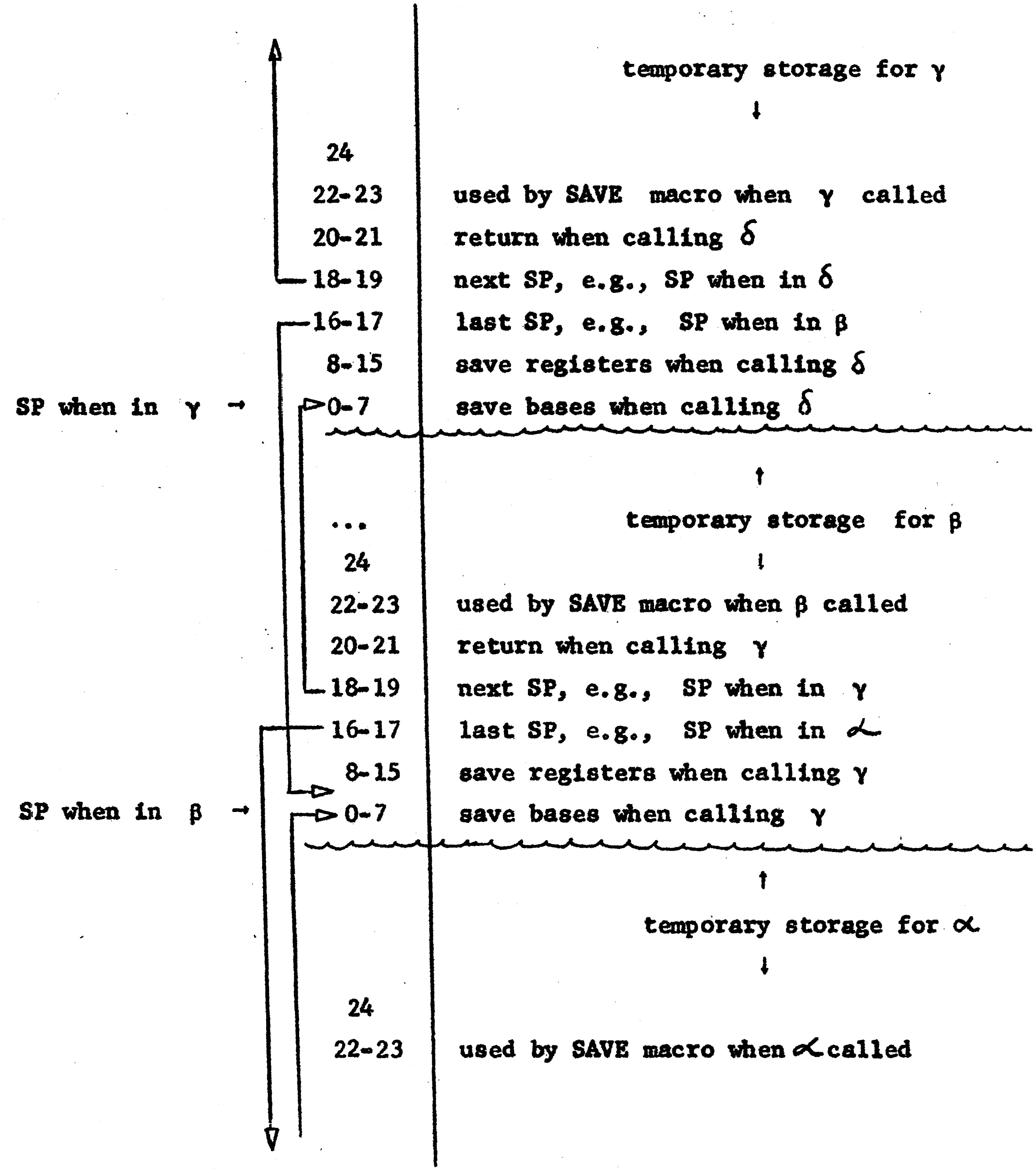
L I N K A G E

	LEFT HALF	RIGHT HALF
	LK.IN	ID=0
	LK.OU	ID
	...	...
LK.IN	INT.N	K
	...	...
INT.N	PS_1	VS_1
	PS_2	VS_2
	PS_3	VS_3
	...	...
Z_1	W_1	FI
	**	M_1
Z_2	W_2	FI
	**	M_2
Z_3	W_3	FI
	**	M_3
	...	...
W_1	Pw_1	Z_1
	$V(EXPR_1)$	W_{1-1}
W_2	Pw_2	Z_2
	$V(EXPR_2)$	W_{2-1}
W_3	Pw_3	Z_3
	$V(EXPR_3)$	W_{3-1}
	...	...
Pw_1	TYPE	W_{N1}
	$SEGN_1$	
	$SYMN_1$	
Pw_2	TYPE	W_{N2}
	$SEGN_2$	
	$SYMN_2$	
	...	...
PS_1	"Internal symbol"	
PS_2	"Internal symbol"	
	...	...
$SEGN_i$	"External segment"	
$SYMN_i$	"External symbol"	

636 System Standard CALL, SAVE, and RETURN Macros

June 14, 1965

The standard CALL, SAVE, and RETURN macros use a stack. The usage of various locations in the stack is shown in the diagram. For the purpose of the diagram α calls β, β calls γ, and γ calls δ.



A. The CALL Macro

The standard CALL is:

```
STCD    SP|20
TRA     ENTRPT
TRA     *+2
EAPAP   AL
```

where ENTRPT is the entry point of the procedure being called and AL is the location of the argument list. In BSA there are three variations of the CALL.

1) No Argument list.

```
CALL    <SEG>|[ENTRY],M
```

expands to:

```
STCD    SP|20
TRA     <SEG>|[ENTRY],M
TRA     *+2
EAPAP   SP|0,*
```

2) Argument list explicitly constructed.

```
CALL    <SEG>|[ENTRY],M(& ALST)
```

expands to:

```
STCD    SP|20
TRA     <SEG>|[ENTRY],M
TRA     *+2
EAPAP   ALST
```

ALST is the location of the first argument in the argument list. The argument list must have been constructed prior to execution of this CALL.

3) Argument list constructed by BSA.

```
CALL  <SEG>|[ENTRY],M(ARG1,?i, ...,ARGn)
```

expands to:

```
EAPBP    ARG1              generate pointer   ┐
STPBP    SP|K              to ARG1            │
LDAQ     AP|2*(i-1)        transfer pointer   │
STAQ     SP|K+2            to ith input arg   ├   prologue
 ...                                          │
EAPBP    ARGn              generate pointer   │
STPBP    SP|K+2*(n-1)      to ARGn            ┘
STCD     SP|20
TRA      <SEG>|[ENTRY],M
TRA      *+2
EAPAP    SP|K
```

The prologue generates the argument list. The generation of the pointers which go in the argument list is the same for all address types except the form ?i, which is merely the transfer of an existing pointer.

B. <u>The SAVE Macro</u>

The standard SAVE is:

```
STRS     SP|8
STB      SP|0
EAPAP    SP|20,*    ┐ set argument pointer
XEC      AP|1       ┘
LDAQ     SP|18      ┐
LDXO     16,DU      │ go to next stack level
ADQ      T,DU       │
STPSP    SP|18,*0   │
EAPSP    SP|18,*    │
STAQ     SP|18      ┘
STCD     SP|22      ┐
LDXO     SP|22      │ set linkage pointer
LBRLP    LB|0,*0    ┘
```

T is the amount of temporary storage which this procedure uses in the stack. The BSA SAVE macro expands into the above sequence. T is known to BSA and is the sum of the explicitly declared stack storage (TEMP and TEMPD pseudo-operations) and the stack storage used for argument lists generated by the prologue.

C. The RETURN Macro

The standard return is:

```
LDB     SP|16,*
LDR     SP|8
RTD     SP|20
```

There are two variations of the RETURN in BSA.

1) Normal return.

```
RETURN
```

expands into the above sequence

2) Return to an optional exit.

```
RETURN  ?n
```

expands to:

```
LDAQ    AP|2*(n-1)
LDB     SP|16,*
STAQ    SP|22
LDR     SP|8
TRA     SP|22,*
```

This variant returns to the caller at a location which was given, at the call, as the nth argument.