MASSACHUSETTS INSTITUTE OF TECHNOLOGY
Electronic Systems Laboratory
COMPUTER-AIDED DESIGN PROJECT
Cambridge, Massachusetts 02139

ESL Memorandum 70429-M-210-0

MEMO TO: Files

FROM: J. E. Rodriguez

DATE: January 6, 1968

SUBJECT: AED in MULTICS

A. INTRODUCTION

This document presents the initial design of an AED compiler for MULTICS.

The main purpose of this initial design is to facilitate a first bootstrap of the AED system from the IBM 7094 to the GE 645.

It is expected that a second bootstrap from the 645 to the 645 will take place shortly after the initial system is running in the MULTICS environment.

The design of the initial compiler and support library is intended to provide an environment which will minimize machine dependent problems on the basis of the AED effort's experience with the UNIVAC 1108 and IBM 360 bootstrap efforts.

Questions of compatibility with MULTICS standards have been considered and it is felt that their final solution should be postponed until the second bootstrap takes place. For the first bootstrap, compatibility is provided only insofar as it is required to guarantee the success of the initial bootstrapping effort. This essential requirement is satisfied by

a. providing means for calling a procedure outside the AED environment from any AED procedure.

b. allowing calls to AED procedures from the outside world only at certain selected points.

B. THE AED ENVIRONMENT

A process containing AED programs consists of:

1. One AED data segment.
2. An arbitrary number of non-AED data segments.
3. An arbitrary number of procedure (AED and non-AED) segments.
4. The necessary linkage and symbol segments to accompany the above.

dp

a. The AED Data Segment

An AED procedure may directly access only data contained in the AED data segment. Data from any other segment must be moved to the AED data segment before an AED procedure can access it. Moving will take place, under normal circumstances, through:

i. storage allocation procedures activated by ft2 faults.

ii. calls in the AED I/O package (IOBCP) or an equivalent substitute.

Data in the AED data segment falls into five categories:

i. Internal static

All variables, constants, and argument lists, in non-recursive AED procedures, are in internal static storage. The storage for these variables is obtained and initialized if necessary at the time that the first call to a procedure of a segment is attempted.

ii. Automatic

Own variables and argument lists in recursive AED procedure are in automatic storage. A chained stack in the AED data segment is maintained by AED support procedures to hold the automatic storage.

iii. External

Separately compiled data segments are moved into the AED data segment at the time that their first reference is attempted.

iv. Common

All common variables are assigned on the AED data segment at the opposite end from all the other storage. The two categories of storage (i.e., common and non-common) share the data segment on a collision basis. Common may not be PRESET.

GPL

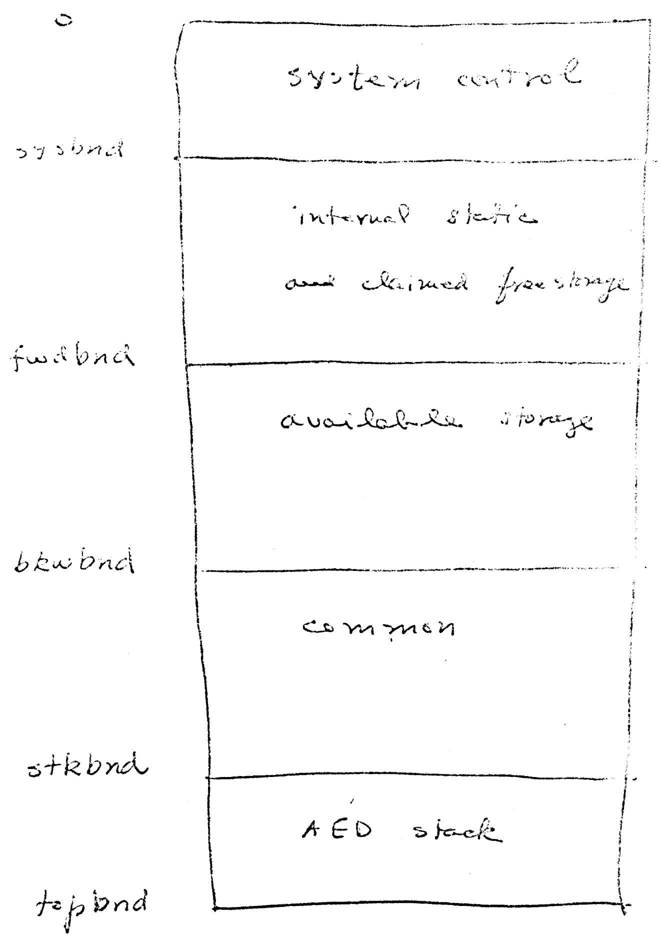


Figure 1a - Layout of AED data segment

v. Free Storage

All storage in the AED data segment not attached to any of the first four classes is administered by the Free Storage package.

Figure 1a shows the layout of the AED data segment. The segment is divided into five regions as follows:

1. - system control [0-sysbnd]
2. - internal static and claimed free storage [sysbnd - fwdbdn]
3. - available storage [fwdbnd - bkwbnd]
4. - common [bkwbnd - stkbnd]
5. - AED stack [stkbnd - topbnd]

The system control region extends from location 0 to location "sysbnd".

The internal static and claimed free storage region extends from "sysbnd" to "fwdbnd". This region grows as "fwdbnd" is increased by the AED storage allocator and/or the free storage system.

The available storage region extends from "fwdbnd" to "bkwbnd". This region shrinks from both ends as "fwdbnd" is increased and "bkwbnd" is decreased.

The common region extends from "bkwbnd" to "combnd". This region grows as "bkwbnd" is decreased by the AED storage allocator.

The AED stack region extends from "combnd" to "topbnd". This region varies as "topbnd" is changed by the enter and leave sequences of AED recursive procedures. The maximum value of "topbnd" is given by "fulbnd".

Address	Name	Description
0 - 7	-	unused
8	sysbnd	address of first location in forwards-growing region
9	fwdbnd	address of next available location in forwards-growing region
10	bkwbnd	address of next available location in backwards-growing region
11	combnd	address of first location in backwards-growing region
12	topnd	address of current AED stack frame
13	fulbnd	address of last location in the data segment
14	dbgptr	address of trace request table
15	-	unused
16-17	condptr	address of "condition" procedure segment
18-19	reverptr	address of "reversion" procedure segment
20-21	unwinder	address of "unwinder" procedure
22-23	aedlnkgptr	address of "aedlnkg" procedure segment
24-31	-	unused
32-50	statistics	array of procedure event statistics
51-(sysbnd-1)	-	unused

Figure 1b - AED Data Segment System Control Information

C. DATA TYPES

Identifiers of type real, integer, boolean and pointer are allocated one word of storage. The principal consequence of the single AED data segment is that pointer variables require only 18 bits to contain an address.

Identifiers of type label, procedure and switch which are used as the operand of a LOC (either explicitly or implicitly when used as arguments or in presets) are allocated six words of storage aligned at an even address.

In data structure declarations, unpacked components are handled more efficiently than any packed component. Among the various forms of packings, left half and right half packing is handled more efficiently than any other form of packing.

D. INTER PROCEDURE COMMUNICATION

The compiler supports two forms of argument lists for inter procedure communication, the AED environment argument list and the MULTICS argument list. The choice of the form of argument list is made at compilation time on an individual procedure basis or as a global option for all procedures in a given compilation. The normal mode is assumed to be the AED environment argument list.

Since the form of the argument list is determined on an individual basis, it is possible to have in the same program any mixture of forms of procedure definitions and procedure calls.

CALL, SAVE and RETURN conventions for both forms of procedures are identical, and consistent with the MULTICS process stack requirement.

AED environment argument list

AED environment argument lists consist of n words where n is the number of arguments. Figure 2 shows the storage structure of the argument list. Each word contains a "typed" short data pointer. The left half of the word contains a relative pointer (offset in the AED data segment) to the datum. The operation field contains a code indicating the type of the datum pointed to by the left half.

The end of the argument list is indicated by a 1 in the high order bit of the operation field.

An empty argument list is signified by a null argument list pointer. Table 1 shows the relationship between AED argument lists and the type of data items being transmitted.

AED pointer data maps into MULTICS relative pointer data. Procedure items differ from the corresponding entry data of MULTICS in that the code 20_8 has been placed in the operation field of the even words of the program point ITS pair. The differentiation between label and procedure items is necessary because an AED procedure with a formal parameter of type procedure may be called with an actual parameter of type label. By looking at the data item, the calling procedure can decide whether to call the procedure directly or call the "unwinder" instead.

MULTICS argument list

A MULTICS argument list consists of n+2 or n+3 word pairs where n is the number of arguments.

Figure 3 shows the storage structure of an argument list. The first word pair contains control information in the even word as follows:

i. the left half is 2*(n+1)

ii. the right half is zero if there are n+2 word pairs and 2 if there are n+3 word pairs.

The next n word pairs are ITS pairs for each of the arguments. Immediately following is an ITS pair for the value of the procedure. This ITS pair appears whether or not the procedure has been declared to be valued in the program originating the call. The arguments and value ITS pairs point to a datum.

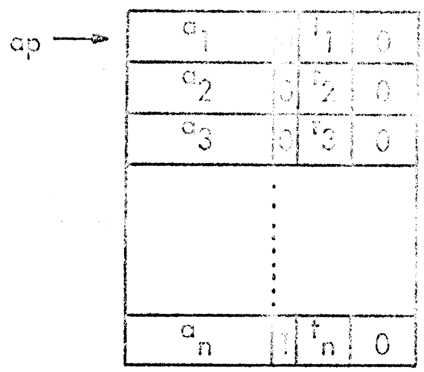


Fig. 2 AED Environment Argument List

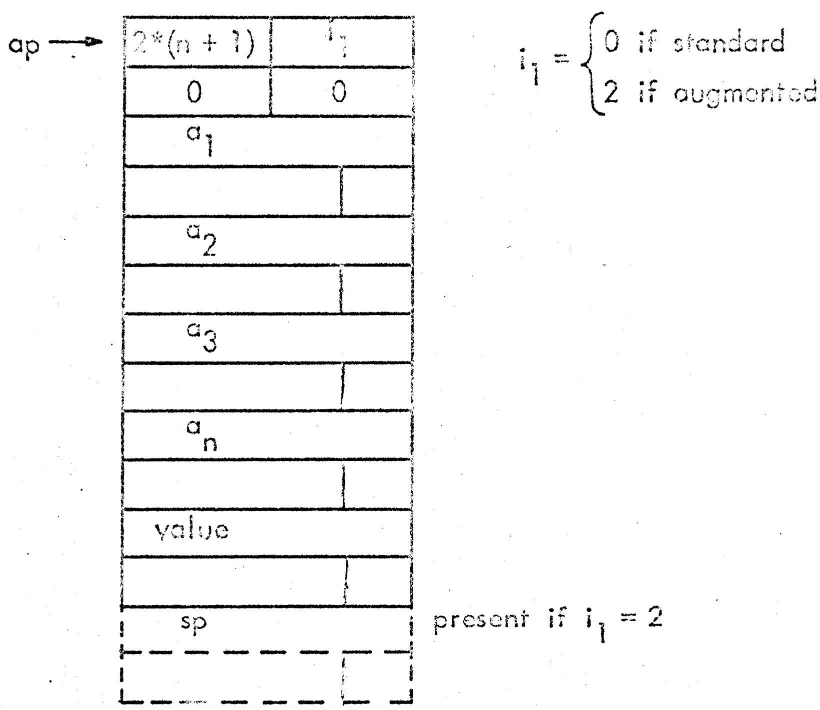


Fig. 3 MULTICS Argument List

Type No.	Argument Type	Storage Structure
1	integer, boolean	
3	real	
14	pointer	
15	label	program point stack pointer
16	procedure	program point
17	1-dimensional array of integer, booleans	
19	1-dimensional array of reals	
30	1-dimensional array of pointers	
31	switch	

TABLE 1

Types of Arguments and Their Storage Structure

Augmented argument lists are used whenever the subject of the call is a parameter. In this case an n + third ITS pair is present and it contains the stack frame pointer for the environment of the called procedure.

<u>The procedure text</u>

An AED procedure consists of:

1. enter sequence
2. initialization sequence
3. body
4. return sequence

This sequence is physically arranged in the order enter sequence, body, return sequence and initialization sequence. Figure 4 shows the skeleton of the procedure text as it may appear in an assembly listing.

```
procname:   ldaq    ent.info
            eax0    entertype
            tsbbp    aedlnkg    aedlnkg
            tra     init
            vfd     18/0,6/6,12/nchar
            aci     'procname'        .C. for procedure name

body:       null
             .
             .
             .

return:     ldq     value             or fld value or null
            eax0    leavetype
            tra      aedlnkg    aedlnkg

init:       null
             .
             .
             .
            tra     body
```

Figure 4. Skeleton of Procedure Text

During execution of an AED procedure the base register pairs are used as follows:

ap - points to the procedure argument list

bp - points to the base address (in the AED data segment) of internal static storage in non-recursive procedure or to the AED stack frame in recursive procedures.

lp - points to the linkage segment.

sp - points to the process stack frame.

Base Register pair bp is loaded in the save sequence from a word pair in the procedure linkage section. The contents of this word pair is set to point to the proper address in the AED data segment at the time the first call to any procedure in the associated text segment takes place.

The four base register pairs are saved at the end of the save sequence in the process stack frame. The procedure text never modifies the contents of bp, sp, and lp. The ap pair is modified only when a call is made. However, the old value is always restored by the return sequence of the called procedure.

1. <u>Enter sequence</u>

The function of the enter sequence is to secure a process stack frame, save whatever registers are necessary in either the old frame or the new frame, and obtain whatever dynamic storage is required.

2. <u>Initialization Sequence</u>

The initialization sequence performs the following three functions:

i. It sets up every usage of an argument in an argument list.

ii. It saves in an internal static location the address of every argument used by an internal procedure.

iii. It sets up the process stack pointer value in every label entry datum used as an argument.

3. <u>Body</u>

The body contains the working code of the procedure.

4. Return sequence

The function of the return sequence is to set the value of the procedure, free up whatever dynamic storage was obtained by the save and initialization sequences and return to the caller.

The in-line code generated for the enter and return sequences is a special subroutine call on an appropriate entry in the AED utility routine segment aedlnkg . There are entries in this segment for handling four types of enter sequences, 12 types of normal returns and two types of abnormal returns. Table 2 shows the code assignment for the aedlnkg routines.

One of the advantages derived from the fact that all procedures enter and leave through common sections of code is a ready-made debugging facility with dynamic tracing and multiple breakpoints at a sufficiently fine level, i.e., a procedure call, for most high level language debugging needs.

An AED compilation contains the following items:

1. For each defined external procedure, the standard MULTICS entry sequence to load the lp base pair and transfer to the procedure entry address.

2. For each external reference, the standard MULTICS link.

3. A link, stat.loc, to aed data [xxxxxx] where xxxxxx is the procedure segment name. This link is initially set up with a trap-before-link to invoke the AED data segment static storage loader, aed datmk [aed datmk] with argument list trap.arg.
   Trap.arg is a two word element residing in the text segment and has the structure shown in Figure 5. The first word contains the total length in words, of the internal static storage in the left half and the length of the common storage in the right half. The second word is a relative pointer to a link which when snapped points to the first block of initialization data.

0	--	enter AED non-recursive procedure
1	--	return from non-valued AED non-recursive procedure
2	--	return from integer, boolean or pointer AED non-recursive procedure
3	--	return from real valued AED non-recursive procedure
4	--	enter AED recursive procedure
5	--	return fron non-valued AED recursive procedure
6	--	return from integer, boolean or pointer AED recursive procedure
7	--	return from real AED recursive procedure
8	--	enter MULTICS non-recursive procedure
9	--	return from non-valued MULTICS non-recursive procedure
10	--	return from integer, boolean or pointer MULTICS non-recursive procedure
11	--	return from real MULTICS non-recursive procedure
12	--	enter MULTICS recursive procedure
13	--	return from non-valued MULTICS recursive procedure
14	--	return from integer, boolean, pointer MULTICS recursive procedure
15	--	return from real MULTICS recursive procedure
16	--	abnormal return through label
17	--	abnormal return through switch
18	--	reserved, see debugging facilities (Section H)

Table 2. Code Assignment for aedlnkg routines

The function of the static storage loader is to obtain storage in the AED data segment for the internal static and, if necessary, for the COMMON storage. After obtaining the storage, the loader processes the initialization data, copying and relocating short pointers as necessary. Links to external references are snapped by the loader before copying them into the data segment.

Upon return to the linker, the fault-inducing link contains an ITS pair pointing to the AED data segment location which will thereafter serve as the addressing base for the internal static storage.

4. For each recursive procedure a word pair in the linkage segment as shown in Figure 6a.

   This word pair, sv.stack, is an ITS pair pointing to the AED stack frame for the last invocation of the procedure. If the procedure is not active, sv.stack is the null pointer.

5. For each procedure defined a word pair in the text segment as shown in Figure 6b.

   This word pair, ent.info, is the argument for the enter sequence routine. The first word contains the offset of the stat.loc word pair in the left half and either zero or the offset of the sv.stack word pair in the right half. The second word contains either zero or the AED stack frame size (i.e., frame header plus automatic storage) in the left half and zero in the right half.

<u>The process stack and the AED stack</u>

Every procedure uses the MULTICS process stack. In addition, recursive procedures use a separate stack, the AED stack, maintained in the AED data segment by routines in the aedlnkg segment.

trap.arg:	intent	coment	
	initdata	0	trap procedure arglist

Figure 5 - Word pair for control of internal static of an AED compilation.

sv.stack:			save stack pointer

a)

ent.info:	stat.loc	sv.stack	enter sequence argument
	fram.size	0	

b)

Figure 6. Word pairs for control of procedures' enter and leave sequences. One for each procedure defined.

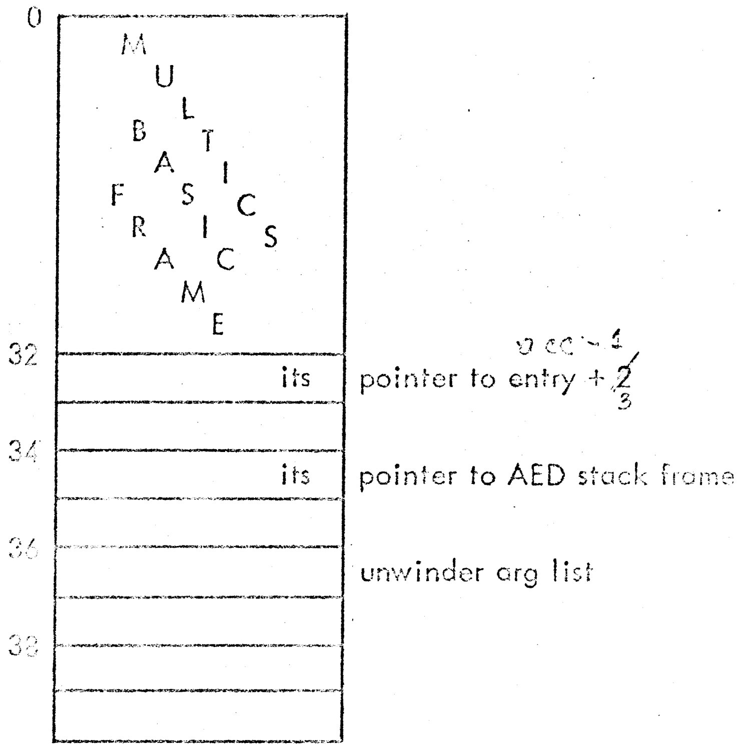


Fig. 7 Process Stack Frame for AED Procedures

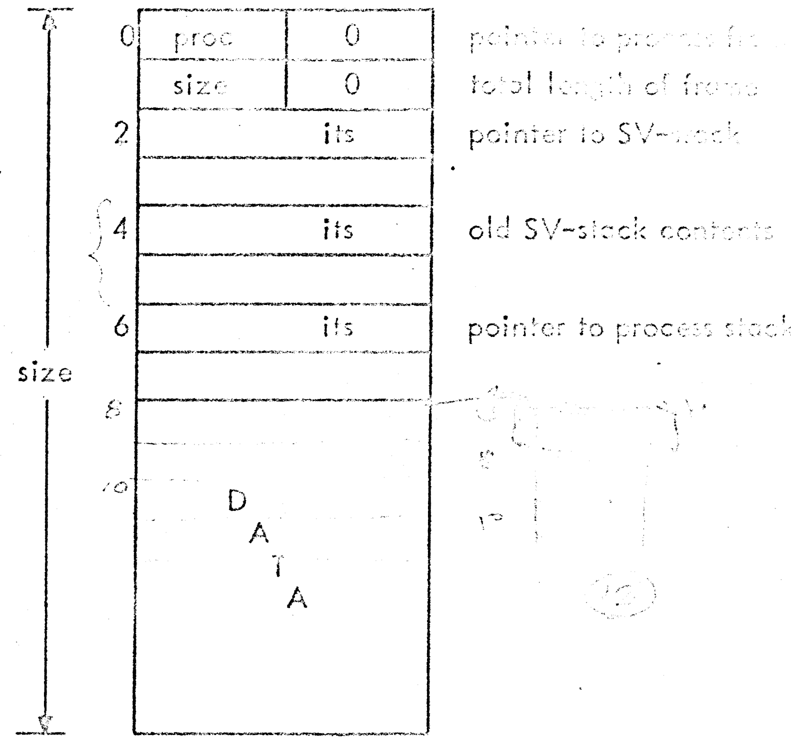
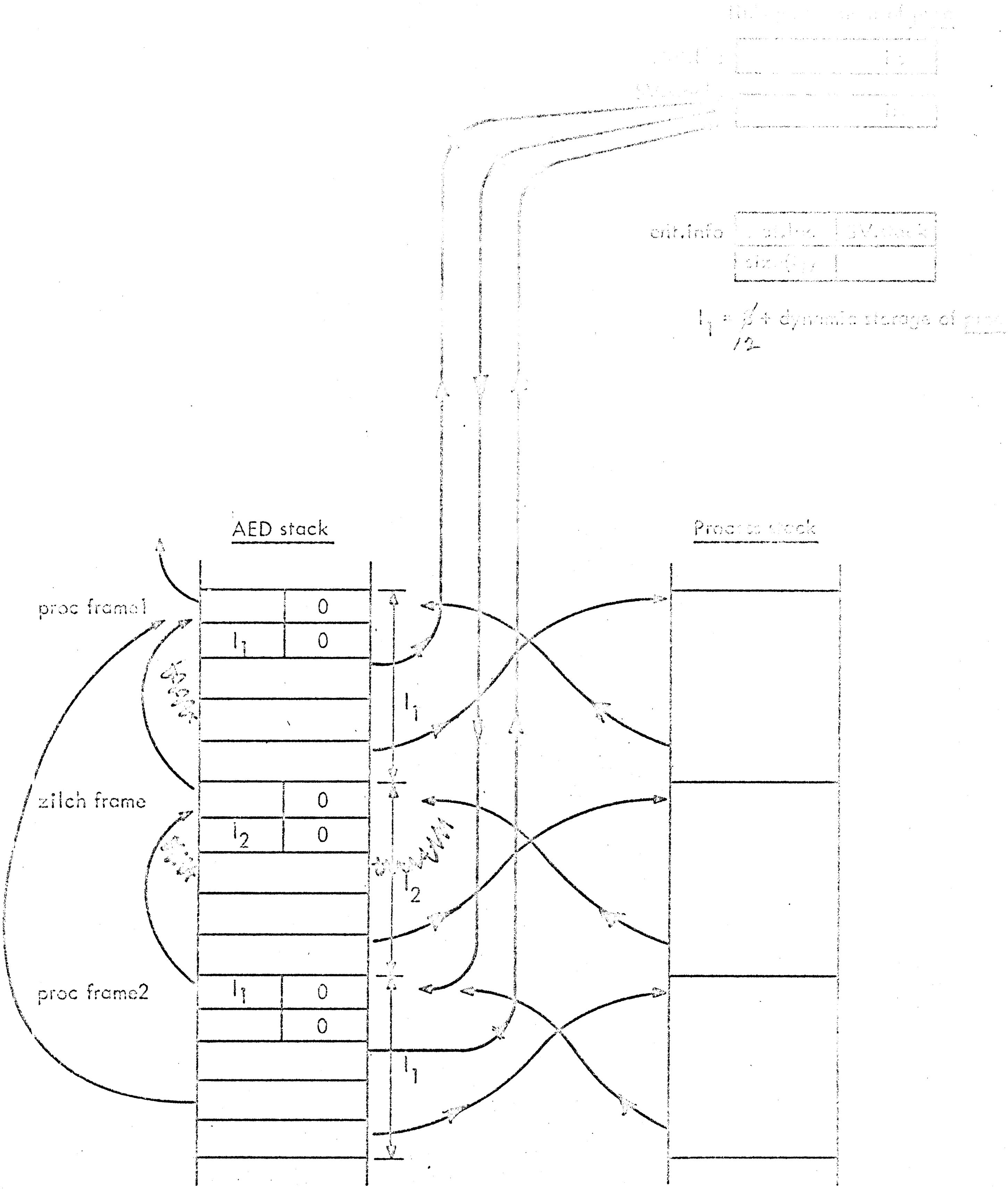


Fig. 8 AED Stack Frame



Note: In general there is not a one-to-one correspondence between AED stack frames and Process stack frames.

Fig. 9 Relationship between Linkage Segment, the AED stack, and the Process stack in Recursive Procedures

The process stack is used to save the machine registers and for temporary storage during enter and leave sequences. A process stack frame procedure is always 40 words long. Figure 7 shows the stack frame layout. The first 32 words are used in the standard MULTICS fashion. Words 32 and 33 are an ITS pair pointing to the location +3 of the procedure which obtained the frame. Words 34 and 35 are an ITS pair pointing, in a recursive procedure, to the AED stack frame associated with the process frame. The last four words, 36 through 39, are used for the "unwinder" argument list in abnormal returns.

The AED stack is used for allocating dynamic storage in recursive procedures. The stack resides in the AED data segment. Each frame in the AED stack consists of a 12-word header followed by the procedure's dynamic storage. Figure 8 shows the layout of the frame. Figure 9 shows the relationship between the linkage segment control word sv.stack, the AED stack and the process stack. The figure depicts the condition existing when a procedure proc is called recursively through the following sequence: proc is called, it calls zilch which in turn calls proc.

Every AED stack frame points to its matching frame in the process stack. Every process stack frame associated with a call on a recursive procedure points to its matching AED stack frame.

The old value of the sv.stack pair as well as a pointer to that linkage segment location are saved in the AED stack frame. Upon return, the old value is put back in the sv.stack pair.

Addressing strategy

This section discusses the addressing of data items in the procedure text. For this purpose, three classes of data items are considered:

i. static data

ii. dynamic data

iii. arguments

The addressing strategy for each of these classes is shown in Tables 3, 4, 5 and 6.

The following conventions are used:

i. for non-recursive procedures, the base pair bb bp points to the zeroth word of internal static storage in the AED data segment.

ii. for recursive procedures bb bp points to the zeroth word of the AED stack frame.

iii. disp denotes the offset of the zeroth word of a data item from the bb bp value.

iv. the base pair ab ap points to the argument list of the procedure.

Common		disp < 16K	unsubscripted	eaxj bb common,* opr bb disp,j
			subscripted	eaxj bb common,*i opr bb disp,j
		disp > 16K	unsubscripted	eaxj disp opr bb common,*j
			subscripted	eaxj disp,i opr bb common,*j
Internal	Recursive	disp < 16K	unsubscripted	eaxj lp stat.loc,* opr bb disp,j
			subscripted	eaxj lp stat.loc,*i opr bb disp,j
		disp ≥ 16K	unsubscripted	eaxj disp eaxj lp stat.loc,*j opr bb 0,j
			subscripted	eaxj disp,i eaxj lp stat.loc,*i opr bb 0, j
	Non-recursive	disp < 16K	unsubscripted	opr bp disp
			subscripted	opr bp disp,i
		disp ≥ 16K	unsubscripted	eaxj disp opr bp 0,j
			subscripted	eaxj disp,i opr bp 0,j
External			unsubscripted	opr lp name,*
			subscripted	opr lp name,*i

TABLE 3

Addressing Strategy for Static Data

Local	label		opr	name
	disp < 16K	unsubscripted	opr	bp disp
		subscripted	opr	bp disp,i
	disp > 16K	unsubscripted	eaxj opr	disp bp 0,j
		subscripted	eaxj opr	disp,i bp 0,j
Global	label		opr	lxxxxxi
	disp < 16K	unsubscripted	eaxj opr	lp sv.stack,* bb disp,j
		subscripted	eaxj opr	lp sv.stack,*i bb disp,j
	disp > 16K	unsubscripted	eaxj opr	disp lp sv.stack,*j
		subscripted	eaxj opr	disp,i lp st.stack,*j
Components				bp disp,i

TABLE 4

Addressing Strategy for Dynamic Data

Non-Program Point	Local	unsubscripted	opr ap argno,*
		subscripted	opr ap argno,*i
	Global	unsubscripted	opr bp argwrd,*
		subscripted	opr bp argwrd,*i
Program Point	Local	procedure	eapbp sp 26,* eapbp bp argno,* opr pxxxx
		label	opr lxxxx
		switch	opr sxxxx
	Global	procedure	eapbp bp argwrd,* opr pxxxx
		label	opr lxxxxg
		switch	opr sxxxxg

Note: argwrd is a location in internal storage containing a copy of the contents of ap argno.

TABLE 5

Addressing Strategy for Arguments

```
pxxxx:   lda     bp  0                      get program point ITS
         ana     =ø20000,dl                 is it a procedure?
         tnz     bp  0,*                    yes, execute
         eax0    16                         no, is a label
         tra      aedlnkg  [aedlnkg]        perform abnormal return


lxxxx:   eapbp   bp  argno,*                get pointer to label datum
         eax0    16                         perform abnormal return
         tra      aedlnkg  [aedlnkg]


lxxxxg:  eapbp   bp  argwrd,*               get pointer to label datum
         eax0    16                         from initialized location
         tra      aedlnkg  [aedlnkg]


sxxxxg:  eapbp   bp  argwrd,*               ...
         eax0    17
         tra      aedlnkg  [aedlnkg]


lxxxxi:  eapbp   bp  labdata                get pointer to internal
         eax0    16                         label datum
         tra      aedlnkg  [aedlnkg]


sxxxxi:  eapbp   bp  labdata                ...
         eax0    17
         tra      aedlnkg  [aedlnkg]
```

TABLE 6

Internal Compiler Generated Routines to Affect Abnormal Returns

E. Preset Data

The initialization of internal static data[1] whose values have been preset at compile time is performed by the trap procedure aed-datmk immediately after it has allocated the storage in the AED data segment.

The bead (ent.info) supplied as an argument to the trap procedure has a component which points indirectly through a link to the beginning of the preset information. The preset information consists of a sequence of variable length blocks. Each block has a header containing control information and a body with the actual preset data. The end of the sequence of blocks is recognized by a word of all zeros where the first word of a header is expected. Figure 10 shows the storage structure of a complete sequence of print information blocks.

The header of a preset block consists of three words. The first word contains the origin of the preset data on the left half and the number of words in the body on the right half. The origin is given as an offset from the beginning of the static data for the procedure segment. The next two words in the header are relocation codes. There is one relocation code for each half word in the body of the block. A variable length coding scheme is used. No relocation is indicated by a code of zero (one bit) all other relocation codes are four bits long. Figure 11 shows the code used.

A brief description of the interpretation of each relocation code as well as instances of their use by the compiler follows.

[1] Neither COMMON nor dynamic storage can be initialized in the GE 645 implementation of AED.

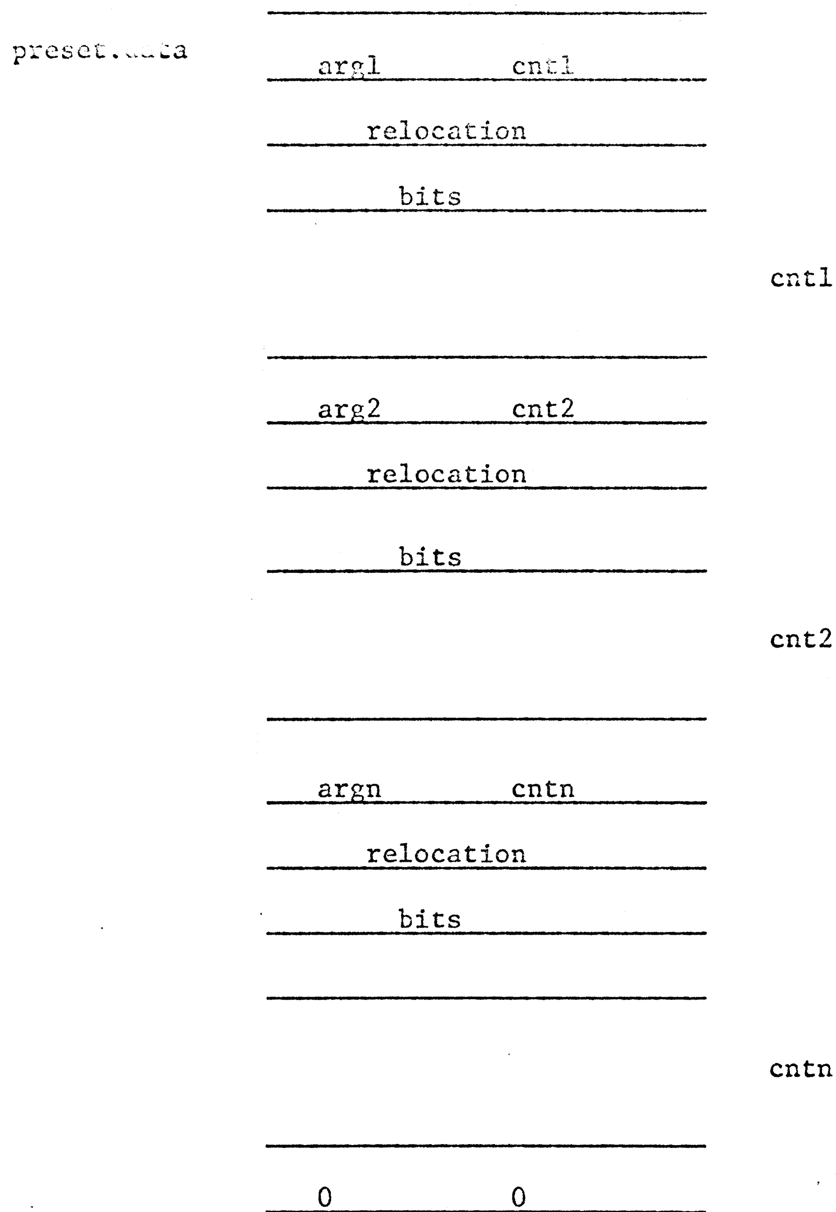


Figure 10 - Storage Structure of Preset Data Information

0	--	absolute
1000	--	add bp
1001	--	add lp
[illegible]	--	snap and use offset
1011	--	subtract from common
1100	--	use bb
1101	--	use lb
1110	--	snap and use segment
1111	--	use pb

Figure 11 - Relocation Codes

0 -- absolute -- the half word is copied into the AED data segment without modification. Used for constants.

1000 -- add bp -- the internal component of the static data pointer (stat.loc) is added to the contents of the half word. Used to preset pointers to static data.

1001 -- add lp -- the internal component of the linkage section address is added to the contents of the half word.

1010 -- snap and use offset -- the half word contains the offset of a link. This link is snapped and the internal component of the effective address is copied into the AED data segment. Used to preset the program point in entry data for procedures and to preset pointers to external data.

1011 -- subtract from COMMON -- the contents of the half word is subtracted from the address of the first location of COMMON storage. Used to preset pointers to variables in COMMON.

1100 -- use bb -- the external component of the static data pointer is copied into the AED data segment. Used to preset the segment number in long pointers to static data.

1101 -- use lb -- the external component of the linkage section address is copied into the AED data segment.

1110 -- snap and use segment -- the half word contains the offset of a link. This link is snapped and the external component of the effective address is copied into the AED data segment. Used in conjunction with "snap and use offset (1010)" to preset the program point in entry data for procedures.

1111 -- use pb -- the external component of the procedure entry address is copied into the AED data segment. Used to preset the program point in entry data for labels and switches.

F. EXTERNAL DATA

Static data of a program can be made available to other programs by name through the use of the EXTERNAL declaration and the PRESET statement.

A program defines an identifier as external data if the identifier is declared to be external and is preset in that program. A program references an identifier as external data if it is only declared to be external. The naming conventions used to define and reference external data are the same as the conventions used to define and reference external procedures (see Section G).

A reference to an external datum is accomplished with two links: a reference link and a definition link. Every procedure which references an external datum has a reference link to that datum in its linkage section. All the reference links to a datum resolve to the definition link for the datum. This definition link is in the linkage section of the procedure where the datum is defined. A reference link has an indirect modifier, while the definition link contains, after it has been resolved, the address of the external datum in the AED data segment. Figure 12 shows the relationship between reference links, definition links, and the AED data segment for an external datum.

To accomplish the definition of an external datum, the unresolved definition link has a trap-before-link procedure associated with it. The trap procedure, aed_datmk [ext_data], has as an argument an element containing:

1. the offset of the stat.loc link in the linkage section
2. the offset of the datum in the procedure's static storage

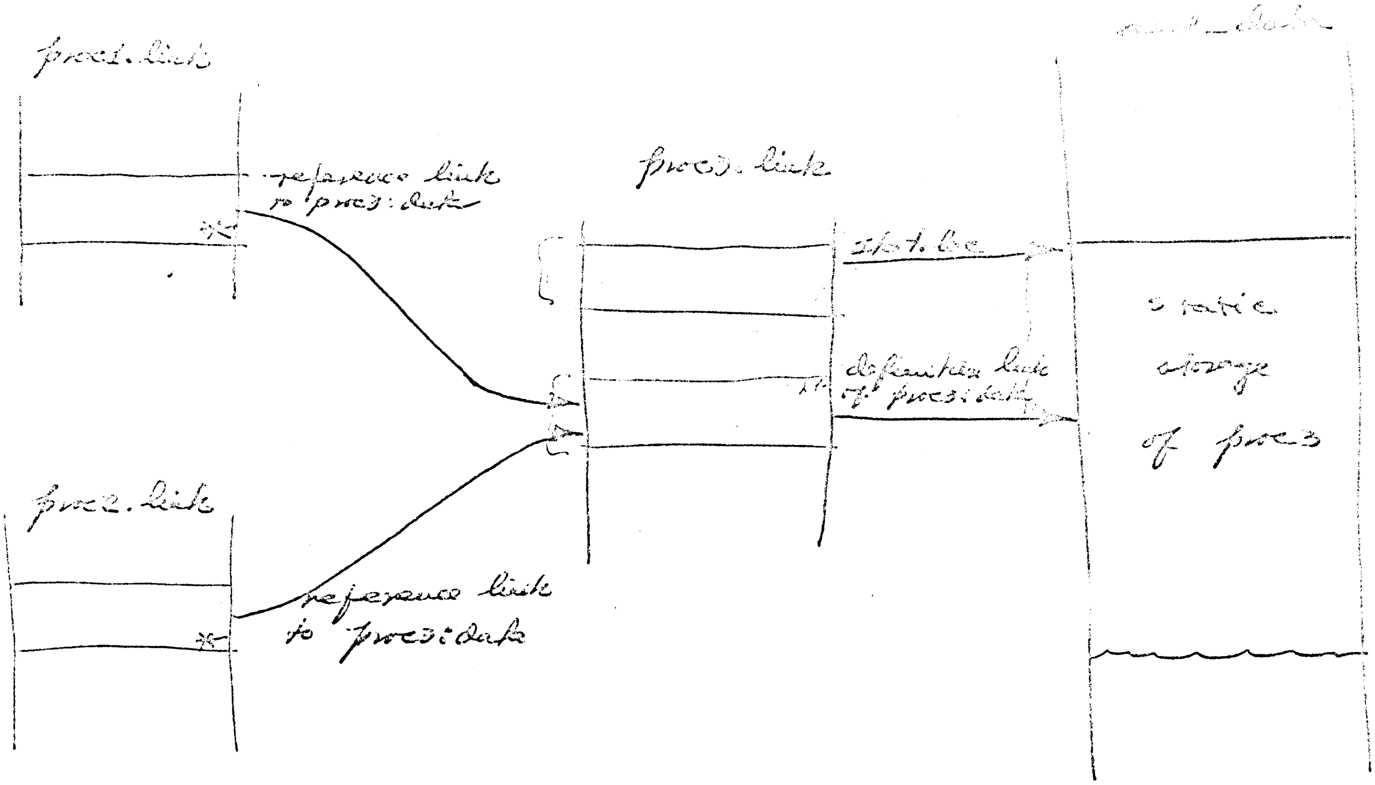


Fig 12.- Relationship between reference links, definition links and AKD data segment for an external datum.

The trap procedure obtains the address of the procedure's static storage from stat.loc, adds to it the datum's offset, and forces the definition link to the computed AED data segment address. It is possible that at the time the first reference to the external datum occurs, the static storage for the procedure has not been allocated. Under these circumstances, the trap procedure aed_datmk [ext_data] defines both the stat.loc link and the definition link. The definition of the stat.loc link implies that the static storage is copied into the AED data segment.

G. NAMING CONVENTIONS FOR EXTERNAL PROCEDURE AND DATA

1. Reference

In order to reference an external procedure or datum, one must specify a segment name and an entry name. To facilitate the specification of these two names the character ":" has been added to the AED character set with the same item building properties as alphabetic characters.

Given an identifier for an external symbol, the following rules are used to derive segment and entry names:

i. If a ":" is not imbedded in the identifier, the segment name and the entry name are the first six characters of the identifier.

ii. If a ":" is imbedded in the identifier, the segment name is the substring composed for all those characters to the left of the first occurrence of a ":" and the entry name is the first six characters to the right of that ":".

Thus if free, setfree, free:free, setfree:setfree and free:setfree are identifiers for external symbols, the corresponding segment and entry names in EPLBSA notation are free [free]. setfre [setfre], free [free], setfree [setfre] and free [setfre].

2. <u>Definition</u>

The segment name of a procedure or external datum defined in a program with file name "alpha" is either "alpha" or the second argument of the command used to invoke the compiler. The first argument of the command is always the file name. If the second argument is given, it is taken as the segment name. Otherwise, the file name is used as the default segment name.

The entry name is derived following the same rules used for external references. Note that if the identifier "proc:name" is used in a definition, the external name of the symbol will not be proc [name] unless the name given to the segment using the rules above is "proc".

H. DEBUGGING FACILITIES

In the procedure linkage machinery, the "aedlnkg" module gains control at entry to and exit from every AED procedure. This facility is used to perform enter and exit time procedure calls on user specified trace procedures exactly as if these calls were inserted at the beginning and end of the user's procedures. The trace procedure to call is determined by searching a user-provided table. Error checking is provided to prevent procedures from attempting to trace themselves.

In order to activate the tracing facility, the user specifies a trace table by a call on procedure aedlnkg:setdbg of the form:

```
aedlnkg:setdbg(trace.table.ptr) $,
```

where:

trace.table.ptr is a pointer to a trace request table constructed by the user.

The effect of this call is to save the trace table pointer in location "dbgptr" in the system control information region of the AED data segment.

The trace request table is a chained list of beads with the structure shown in Figure 13.

nextbd	tracer
procname	0

Figure 13 -- Structure of Trace Request Table Bead

NEXTBD is a pointer to the next bead in the chain. A zero is used to indicate the end of the chain.

TRACER is a pointer to an entry datum for the procedure to be called, i.e., the trace procedure.

PROCNAME is a .C. pointer to the name of the procedure to be traced. This .C. is compared with the .C. placed by the compiler in the body of a procedure text.

The value zero in the procname and tracer fields of a trace request table bead is a special case.

The "aedlnkg" routines search the trace request table as follows:

The procname field of the current request bead is examined. If it is zero then it is assumed that a trace is desired independently of the true name of the procedure. Otherwise, the contents of procname is taken as a .C. pointer. This .C. is compared with the .C. supplied in the procedure text. If the comparison is unsuccessful, the search advances to the next request bead in the table and the tests are repeated.

Once it had been determined that a trace is desired, the tracer field of the request bead is examined. If it is zero, then no trace is effected. Otherwise, the specified tracing procedure is called.

The effects of this search procedure are:

1. a zero procname and non-zero tracer will trace all procedure calls.

2. a non-zero procname and zero tracer will inhibit any trace that may be requested by a request bead located after this one in the trace table.

Trace request tables are constructed using PRESET statements.

The trace procedure specified in the "tracer" component of a trace request table bead is called as follows:

TRACER(name, RETRN, code, val) $,

wheree:

name — is a .C. pointer to the name of the procedure being entered, left, or passed through (in abnormal returns).

RETRN — is a label which can be used to obtain the arguments of the traced procedure using the ISARG package.

code — is an integer specifying the event. The possible values of code are given in Table 2. A code of 18 is used to signal the event "passing through in an abnormal return".

val — is the value of the traced procedure and is meaningful only when "code" indicates a normal return of a valued procedure.

I. SPECIFICATION OF AED AND MULTICS PROCEDURES WITH THE BOOTSTRAP COMPILER

Section D indicates that the compiler supports two forms of argument lists: a standard MULTICS argument list and an abbreviated AED argument list.

The specification of the form of the argument list expected by a procedure occurs at compilation time through an option in the command line. The normal case assumed by the compiler is the abbreviated AED argument list.

The general form of the command line is:

```
                                (AED)
R AED145 filel - segnam -       (MLTX)  -file2-
```

where:

AED145 is the name of the bootstrap compiler.

filel is the primary name of the file to be compiled.

segnam is the optional name of the segment to be created. If "segnam" is omitted, filel is used as the segment name.

(AED) and (MLTX) are the names of the options controlling the form of the argument list.

(AED) means that all procedures defined or referenced in "filel" are AED procedures.

(MLTX) means that all procedures are MULTICS procedures.

file2 if present is the name of a file with secondary name "ALGOL" containing the names of procedures which are exceptions to the rule given by the preceeding option.

"file2" must be line marked with one name per line starting in column 1 and no extra characters (except for the CTSS null character). "file2" is ignored unless "(AED)" or "(MLTX)" is present in the command line.

Examples:

1. R AED145 filel segnam

or

R AED145 filel segnam (AED)

means compile "filel ALGOL" using AED argument lists for all procedures and name the resulting segment "segnam"

2. R AED145 filel segnam (MLTX) file2

means compile "filel ALGOL" using MULTICS argument lists for all procedures except those whose names appear in "file2 ALGOL"

A successful compilation yields a file "filel eplbsa".