M0087

A GUIDE TO MULTICS FOR SUBSYSTEM WRITERS

Chapter II

Intersegment Linking

Elliott I. Organick

Draft No. 3

February 1968

Project MAC

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

LIST OF TABLES

CHAPTER II

INTERSEGMENT LINKING

2.1 INTRODUCTION

One of the key problems needing solution in any programming system is this: Given two segments, < a > and < b> of a process, suppose < a> is a procedure that needs to fetch from, store into, transfer control to or return to < b> . How is <a > "told" where <b > is located in memory so that such intersegment references can in fact be executed?

In an ordinary batch operating system, all the segments of a process are declared, in advance, in some way. All segments are loaded for execution at the outset, and in the loading process, each procedure segment is "told", by establishing its transfer vector, where each of the other segments it needs to know about is located. This "telling" process is known as "linking".

In the Multics environment a process, at the time it is activated for execution, does not provide the supervisory system with an explicit list of the segments that belong to it. The "withholding" of this information is, in a sense, deliberate. The supervisor learns that a certain segment is part of a given process only at the instant that the active process reaches a point in its execution where it needs to have said segment in core memory. Prior to this instant the heretofore unloaded segment is simply a file stored somewhere within the secondary storage hierarchy, under the thumb of the file system.* The supervisor is automatically invoked at the instant the need for each new segment occurs. At this point the supervisor's job is to go get that file, wherever it is, place it in core somewhere, register it as another bona fide segment of the process in question, and "tell" the asking segment where the newly loaded segment has been placed. This, in the broadest

*The "file system" is that part of the supervisory software which organizes, manages, stores and retrieves information from the secondary storage hierarchy.

sense is what is meant by linking in Multics. Bear in mind that the ordinary user will be completely unaware that this interplay between his procedures and the supervisor is going on. Such a grand plan automatically alters our concept of core memory. Its physical dimensions may be fixed, to 2^{18} words, for example, but, under the new concept, its effective size, as far as a user is concerned, is really the sum of core memory (less the space occupied by the supervisory system) and that of the entire secondary store that is managed by the file system.

You would be correct in now sensing that in the Multics operating environment the problems of linking are more complex than in a batch system - but of course - far more operating flexibility is achieved.

2.2 THREE OBJECTIVES OF MULTICS

Three different operating objectives of Multics influence the design of the linking mechanism. We shall state these objectives, see how each affects the process of linking, and then see how this aspect of the linking process is handled in Multics. The three objectives are:

2.2.1 Segment Reloading

When an operating process i is interrupted, the space used for the pages or page tables for some of its segments may be used by some other process j, which may take over the processor. When process i resumes operation, some of its segments, pages or even page tables may be missing. They must be reloaded as needed. Moreover, in order to gain efficiency in the use of core memory, and to reduce core swapping overhead, the new core location of a reloaded page or page table of a segment should not have to be located in the core block corresponding to its last core residence address.

There should be no penalty for the privilege of relocating a segment. In particular, the relocation of a (reloaded) segment should not result in a need to revise any previously established intersegment address references.*

2.2.2 Sharing In-Core Procedures

Certain procedure segments will be in common use in many user processes. Among these are a group of supervisory routines, library subroutines, compilers and assemblers. It should be possible for several processes to share the same in-core copy of a procedure in order to conserve memory space to reduce unnecessary core-to-secondary storage transfers. A first prerequisite for such shared procedures is that it necessarily be a pure procedure. We define a pure procedure as one that does not change (not one bit of it) as a result of being used, i.e., no moving or replaceable parts. The data with which, or on which, such a shared procedure operates will necessarily be different for each process that the shared procedure is attempting to serve. Consequently, the effect of data pointers must somehow change each time the procedure finds itself serving in a different process. A Multics mechanism must be and has been developed to achieve this capability at an overall minimum cost in execution time or storage requirements.

2.2.3 Loading Segments as Needed

When a process begins functioning it should not have to acquire core copies of any more of its segments, or pages of segments,† than is absolutely necessary to begin running. As the process executes, segments

*Strictly speaking this objective is automatically achieved by the segment and page management modules of the supervisor, which are units of the basic file system. We mention this objective in connection with linking only to round out the full picture for the reader.

† To simplify the exposition in this chapter we shall henceforth refer to segments only, but you should keep in mind that pages and the paging mechanism are always implied as a further detail.

should be brought into core or allocated in core only on an as-needed basis. Motivation: In a Multics operating environment a process may have space for most any (or all) of its in-core segments pre-empted frequently, and in most cases, unpredictably.

Therefore, there is little point to undertake the "expense" of loading a given segment unless there is some significant expectation that that segment will be used during the time slice alloted to that process.

Moreover - and more striking - some segments of a process cannot conceivably be known about and therefore loaded in advance. Thus, a user may type at the console the name of a segment as part of a command. Only at that point can the computer learn that the given segment is related to the user's process.

When we speak about a segment brought in as needed, we mean at the time the first executed reference is made. Let's, for example, see what this means for a procedure segment <a> that refers to places in two different data segments, <r>, and <s> and to an entry in one procedure <t>. Figure 2-1 shows a schematic of <a> showing four different instructions as they are likely to be written by the programmer in some symbolic language — we have selected assembly language, but we could have also illustrated with a PL/I or MAD segment.

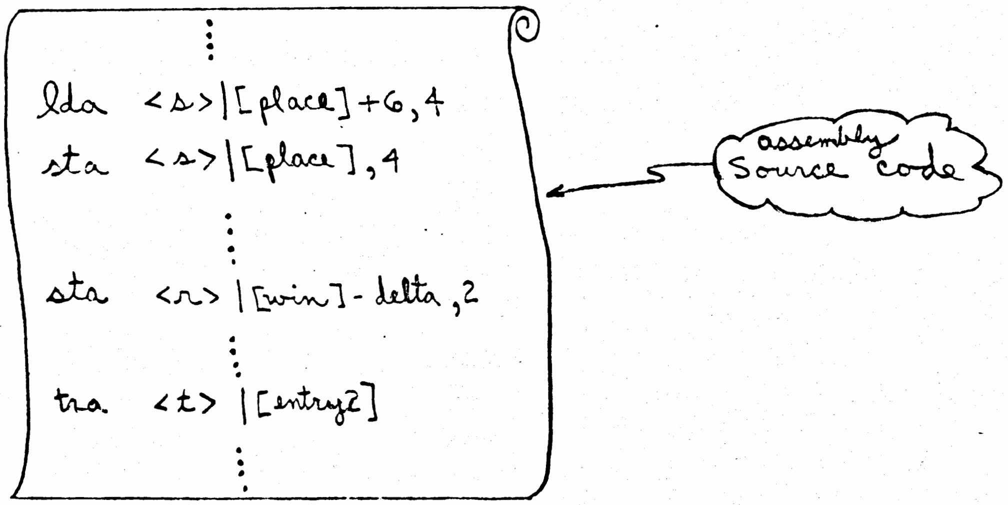


Figure 2-1. Procedure Segment <a> With Four Instructions Making Intersegment References.

Figure 2-1 tells us that < a > might at some time need to know where < s >, < r >, and < t > are each located in core. But, we certainly can't say at the time < a > is loaded when (during the execution of < a >), or in what order < a > will need to know these core locations. Depending on the detailed logic in < a > it's possible that none of these four intersegment instructions will be needed for some run of the process that < a > is serving. It is in this sense that we speak of loading segments on an as needed basis. Thus if in attempting to execute the instruction in < a >:

lda < s > | [place] + 6, 4

a Multics process discovers that the pointer to < s > doesn't exist in its descriptor segment*, then and only then, will < s > need to be loaded (or space for < s > allocated) in core. This concept has been referred to as "dynamic linking". With respect to our particular example of the lda instruction we must bear in mind that at the time < a > is assembled, no one can in general know any of the following vital facts:

1. Where < s > would be variously located in core — and more importantly, what value will have been given to s# in the process we are speaking — that is to say, where in the descriptor segment a pointer to the core location for < s > will eventually be placed.

2. Where, relative to the top of < s >, may [place] be found, i.e., the value of place. The programmer of < a > in general does not even want to be able to know this information about < s >. He only wants to be able to know and be able to use a character string that is the symbolic location for said position internal to < s >.

What mechanism can be employed to complete such an intersegment reference at execution time? Multics provides an automatic mechanism. It's rather complex in detail but not in overall concept.

*By searching a so-called Known Segment Table which is a list of all segment names currently known to belong to this process. The segment number associated with the segment name is found in this table.

To understand this concept we refer to the familiar apparatus known as a symbol table. Any Multics segment that has locations within it which have symbolic names, will have associated with it a symbol table*. Normally this table is prepared automatically by the assembler or compiler from the source language representation of the segment. This table has a standard format, hence, it can be searched in a predetermined way. A successful search of it will locate the numerical equivalent of any local symbol that may have been referred to by another segment. In our example, we would say that associated with < s > is a symbol table which can be searched for the locally unique symbol "place" and its corresponding numerical equivalent.

The linking mechanism can now be seen conceptually as following these basic steps relative to the example

```
lda < s > | [place] + 6, 4
```

1. Determine s# by the following general mechanism:
   Find and load the missing segment < s > from secondary storage, or, if < s > is merely to be a data area in which data is to be generated, allocate the core space for < s >. In doing this, create and add another descriptor word to the descriptor segment for the process. The descriptor word will contain a pointer to < s >. Now s# is determined as the index of the newly-formed descriptor word. At the same time enter in a table called the Known Segment Table the name < s > and its number s#.

2. Determine place, i.e. the value of [place] :
   This can now be done because when < s > is loaded, we will also have loaded its symbol table (and we now know where < s > is located), so we can search the table for place and thence find its numerical value.

*The name used in Multics for this table is the "external symbol definitions". See BD. 7. 01 for details. It's not to be confused with another, more extensive symbol table, called the "Segment Symbol Table" described in BD. 1. 00.

Having determined s# and place, we can complete the process of generating the required machine code and use it to replace the equivalent symbolic form of the instruction

```
lda <s> | [place] + 6,4
```

which was encountered when executing in <a>.

Note the instruction

```
sta <s> | [here] , 4
```

found in Figure 2-1, which is pictured as immediately following the lda instruction. When the lda is executed for the first time, the linage mechasism as we've just described it will have resulted in the loading of <s> and its symbol table into core. So, when the computer next attempts to execute the sta instruction, <s> is already loaded. The job of completing the machine code equivalent of the sta is now quite a bit shorter, because no segment loading is involved. We determine s# simply by searching for and finding it in the Known Segment Table for the process (we just put s# there while loading <s>, you will recall). We then determine here by searching for it, as we did for place, in the symbol table belonging to <s>.

2.3 LINKING DETAILS

Needless to say, a great deal of detail has been skirted in the mechanism we have just described. Some subsystem writers may need to understand this process in some detail in order to provide Multics with the data it needs and in the proper format so that this automatic linking process can be achieved. The subsystem writer who may especially need to know these details is the one who will be building his own processor like an ALGOL or MAD compiler — or an assembler — which will output target code directly; i.e., a processor that does not output code or data in the syntax of a standard Multics-provided language like PL/I, or EPLBSA (assembly language).

We intend to go over more of these details in the remaining sections of this chapter; however, for the present we list in Table 2-1 the key reference sections of MSPM on which the following material is based.

TABLE 2-1

References on Intersegment Linking

Document No.	Title	Remarks
BD. 7. 01	Linkage Section	Primary reference on linking
BD. 2. 01	Binding Info & Format	Secondary information
BX. 14. 01	The Binder	
BD. 7. 04	The Linker	Secondary information
BD. 7. 05	Combined Linkage Segments	Secondary information

We shall proceed by looking at linking details that are made necessary to meet each of the three Multics objectives stated in our introductory discussion. The objectives were deliberately ordered according to the complexity (increasing order) of the explanations that are required.

2.4 RELOCATING SEGMENT < s > AFTER LINKING

We again refer to our example in Figure 2-1. Suppose that sometime after having once established the values s#, place, and here, the space for < s > has been pre-empted for use in another process.* The machine code equivalent to

*Actually, even the same process could require temporarily the space occupied by < s >. If < s > is needed again, it would be presumed that space in core could subsequently be made for it when actually needed.

```
lda <s> | [place] + 6,4
```

will have already been "established".[*] We might look at this equivalent machine code, for the moment in a sort of pseudo symbolic form as

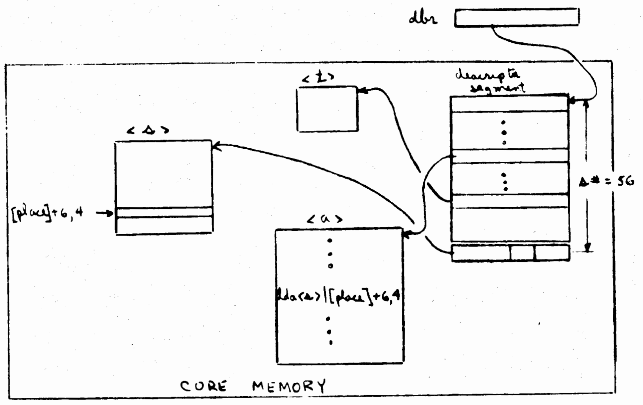
```
lda s# | place + 6,4
           ↑
         value of [place] in <s>
```

Notice that if we later reload < s > into a different core location, as long as we don't change s#, there is no need to further alter our established intersegment reference. This idea is captured pictorially in Figure 2-2.

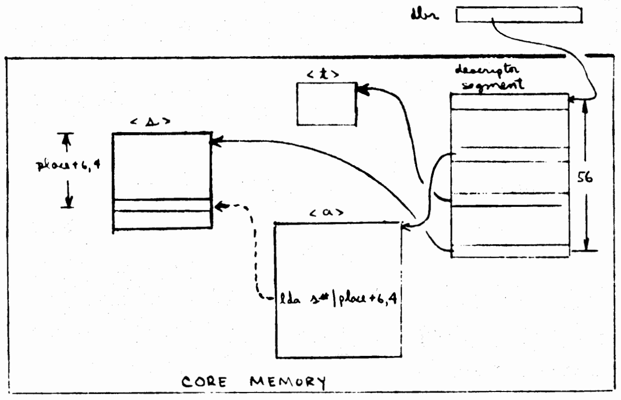
Multics manages to retain the same value s# for < s >, each time < s > is loaded, in a very simple way. When < s > is first loaded a descriptor word for it is created and added to the current end of the description segment. Let's say at position 56. This means that s# = 56. If < s > is ever removed from core memory temporarily, certain bits (33-35) of the descriptor word for < s > (see Table 1-1) are automatically revised to indicate the segment is absent. Any future attempt to address a word in this segment will incur a "directed fault" during the formation of this address. Let's for the moment skip over the details of how the GE 645 and the Multics software handle this fault[†] — we'll pick this up later — and now look only at the resulting effect. As a result of this fault, segment < s > will be reloaded into memory, this time probably into some other core location. When loading is completed, we don't add a new descriptor word for the newly-loaded < s >. Instead, we adjust < s >'s old descriptor word (number 56) to reflect the new core address of < s > (i.e., the "A_B" field). Also, segment missing bits (33-35) are reset to designate (a) that the segment is again

[*] What we mean by "established" will become clear later.

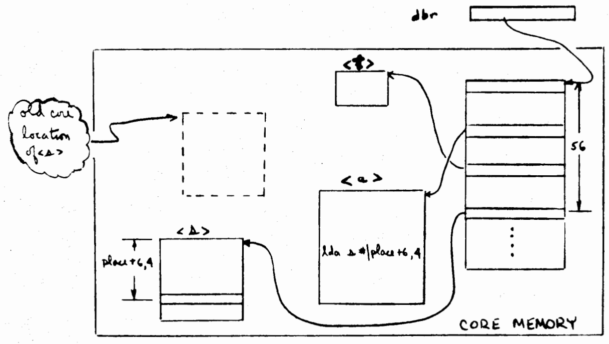
[†] A directed fault is like an IBM 7094 trapping process — like floating point overflow trap. Upon sensing the directed fault condition the GE 645 processor is trapped to a location where certain desired action is involved.



(a) Core memory immediately after < s > has been loaded. New descriptor has been added to point to < s >. Address of lda instruction in < a > not yet "established."



(b) Same as above after establishing the address of the lda instruction. We show it in pseudo symbolic form to suggest that s# and place are now known. Dashed connecting line to < s > | [place] is to show that a connection has in some sense been established.



(c) Same as above after reloading < s > into new location. Note there is no change in the lda instruction of < a >.

Figure 2-2. Loading and Relocating < s > in Core Memory.

present and (b), denote the segment class. (See Section 1.2.9 for an explanation of Segment Class.) Address formation, interrupted by recognition of the directed fault, is now allowed to go to completion as if the fault had not occurred at all.

In summary, the important concept to take note of is that instructions in < a > that refer to locations within < s > do not themselves need to be changed, once "established" in the course of being executed a first time. Each such address is established by determining two values which, generically speaking, are s# and sloc. Sloc is an offset from word zero of < s>. Neither of these values will have changed in spite of the fact that the core location for < s > | 0 may have changed.

2.5 PROCESSES SHARING PROCEDURE SEGMENTS

In the immediately preceding discussion we have been deliberately vague, when discussing the concept of "establishing" an address for referencing a place in another segment. For example, we did not actually give the specific details of the original symbolic reference < s >|[place] as it is found in < a > at the time < a > is originally loaded. Nor did we show the actual format of the numerical equivalent when that numerical equivalent of < s> |[place] was determined.

In this section we will begin to develop these details, which are somewhat complex. Strong motivation for the complexity comes from attempting to satisfy the second of our major objectives, namely insisting that it should in principle be possible for any procedure to be (simultaneously) shared by two or more processes.

We begin by examining Figure 2-3. Here we are reminded that a single GE 645 instruction in one procedure segment, can indirectly reference a location within another segment via a point within an intermediate segment containing an indirect word pair (its pair). Shortly, we will see why it is a good idea to call this middleman segment a "linkage segment". In particular, a single instruction in segment < a > can make a data or instruction (transfer or return) reference to any point within segment < s >, via an intermediate linkage segment. In Multics, every procedure segment that makes such cross references will have an associated linkage segment. Generally speaking, segment < a > has associated with it a linkage segment called < a. link >. (Thus if the name of the segment is "cosine" then the name of its linkage segment is "cosine. link".) Note too that the notation for segment numbers naturally leads to interpreting a. link# as the segment number for < a. link >.

Initially it is hard to see why one needs to bother going to the trouble of making the reference from < a > to < s > an indirect one when it's perfectly feasible to use GE 645 instructions that make direct intersegment references. Direct intersegment referencing was illustrated in Figure 1-12 (b). In the next paragraphs we hope to provide the why of the linkage segment. Without doubt, there are few concepts more crucial to the power of Multics than that of the linkage segment.

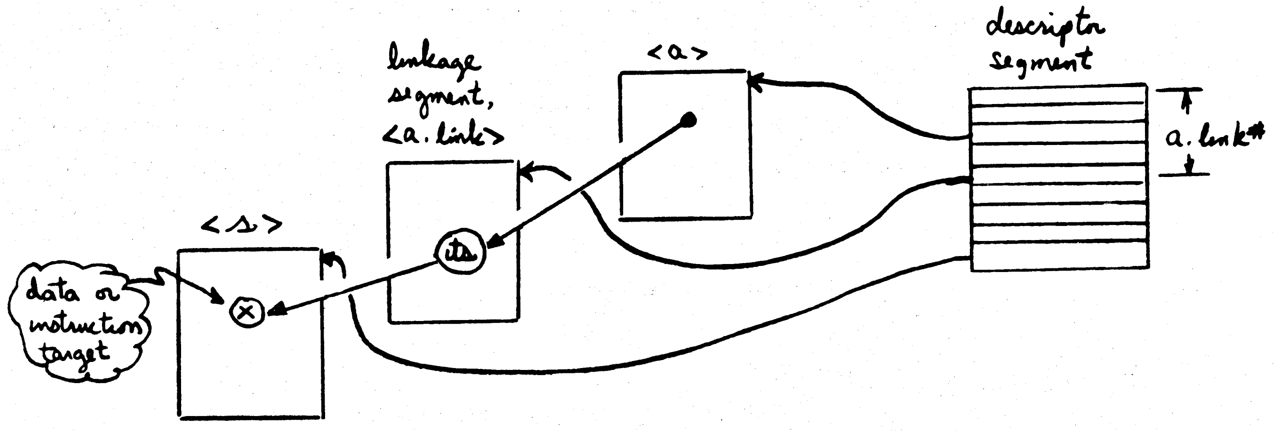


Figure 2-3. Referencing < s > from < a > via an Intermediate Linkage Segment called < a. link >

In Figure 2-4 we imagine that segment < a > is a procedure segment that is currently shared by two processes. Let's imagine that < a > is an assembler or a compiler. An assembler that's being shared by two active processes* may have to deal with two completely different sets of data stored in core memory at the same time.

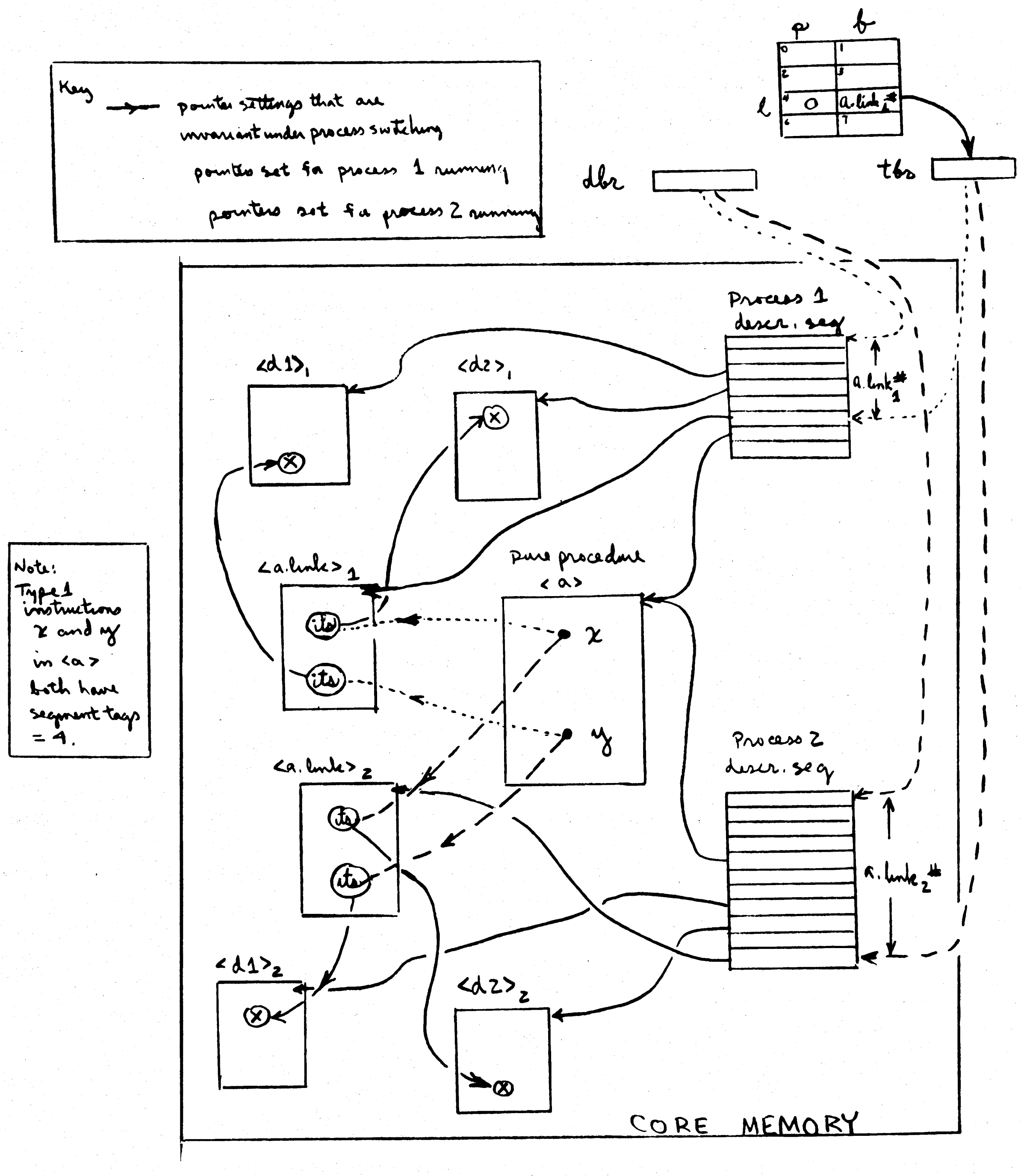
It is highly desirable that < a > be a pure procedure. If so, then, as "ownership" of the processor is switched back and forth between two (or more) processes, the one copy of < a > can function effectively on the two different source language programs that < a > treats as data.

For the assembler to be pure, all data on which it operates, e.g., source language program, symbol tables that are being constructed or being sorted, switch settings, etc., must be maintained in one or more separate segments. Hence there must be a different set of such separate data segments for <u>each process</u> that the assembler currently serves.

Connection to the data segments is made indirectly via the linkage segment of the procedure being executed. It is important to bear in mind that the linkage segment < a. link > is (and must be) generated by the assembler or compiler while it is generating < a >. During execution, the linkage segment is reached from < a > in a standard fashion, namely: one pair of base registers, in particular the lb←lp pair, is <u>always</u> set to point at the associated linkage segment. (E. g., if process 1 is in control, then at the time < a > is called, we imagine that lb←lp is then set to point at the < a. link > in that process. For process 1 we call the linkage segment $<a.link>_1$ and its segment number is called $a.link_1\#$.)

Located within <a. link> will be found the its pairs, one for every unique symbolic intersegment data (or procedure) reference, that appears in the programmer's source code for < a >. But, make a mental note of this fact: the compiler or assembler that's generating < a > and < a. link > cannot

* An "active process" is a technical term in MSPM. It has specified meaning and is defined technically in BJ. 2. 01. A semi-technical definition would be that the supervisory system of Multics knows this process by virtue of it being entered on a list of currently active (running or ready) processes. Each entry in this list gives, among other things, the address of the descriptor segment for that process.


Key
pointer settings that are invariant under process switching
pointers set for process 1 running
pointers set for process 2 running
p
b
l
0
a.link$_i$#
dbr
tbr
Process 1 descr. seg
a.link$_1$#
<d1>$_1$
<d2>$_1$
<a.link>$_1$
pure procedure <a>
x
y
Note: Type 1 instructions x and y in <a> both have segment tags = 4.
Process 2 descr. seg
a.link$_2$#
<a.link>$_2$
its
<d1>$_2$
<d2>$_2$
CORE MEMORY

Processes (i=1, 2) share a common procedure <a> which makes data references to $<d1>_1$, and $<d2>_1$, when functioning for process 1; but makes data references to $<d1>_2$ and $<d2>_2$ when functioning for process 2. Each time process i takes charge, the lb base register is set to a. link_i#. Solid lines represent pointers that do not change as a result of process switching.

Figure 2-4. Two Processes Sharing a Common Reference Procedure

produce the desired its pairs because it doesn't have all the necessary information. It does, however, produce equivalent word pairs which can eventually be transformed into the needed its pairs.

We'll now sketch how this linking mechanism works in one simple case (and look at the actual detail much more closely in the next section). Suppose an instruction in < a > originally appeared in source language as

```
lda <s>|[place] + 6,4
```

The <u>assembled</u> instruction would turn out to be an indirectly addressed lda instruction to a location in < a. link >. The assembled binary machine instruction would if "unassembled", have the form

```
lda lp| k, *
```

Its meaning is as follows:

The segment tag of this instruction is 4. It points to the lb←lp pair. So, the effective pointer is the contents of the lb base register, which presumably has been set previously to a. link#.

The effective internal address is k + (contents of the lp base register). To keep things simple, we have shown in Figure 2-4 that the contents of lp is zero. Here, k is a number determined by the assembler that assembled < a > to point to a strategically located pair of indirect words within < a. link >. This indirect pair will ultimately have the its pair appearance:

s #	0	its
place + 6	0	4

} a completed "link"

when linking is completed.

The format of the linkage segment is the subject of the next section of this chapter, so we need not go into it in any detail here. We will see there how the constant k used in the lda instruction above is determined so that we are sure to address the right pair in < a. link >. Also, we'll see just what the indirect word pair looked like before it gets converted to the desired its pair. We'll also see <u>how</u> it gets converted to the desired its pair.

To take stock — after our short look ahead into the next section — we can summarize what we've said to this point as follows: While operating in process 1, each time an intersegment reference must be made from < a > to some data or procedure segment, an indirect reference is used employing an its pair located in $<$ a. link $>_1$. All external references from < a >, regardless of the destination segment, are routed via its (or itb) pairs located in $<$ a. link $>_1$. This linkage segment acts like a big switch box. Its purpose is to eventually hold its (or itb) pairs which will provide the generalized addresses [segment number, internal address values (or base, internal address values)] corresponding to each unique symbolic data and instruction reference that is given in the original source language coding of < a >.

We now make an interesting observation. When the processor is switched to process 2, < a > is immediately ready to serve in this process, provided there has been loaded an appropriate linkage segment, $<$ a. link $>_2$, to act in a fashion similar to $<$ a. link $>_1$.

How about the problem of being sure lb←lp points to the right a. link# ? Even this is handled automatically, if, for example, we are switching back and forth between two processes that are both busy executing in < a >. Here's how. Suppose we are switching out of process 2 while executing in < a >. In this case the value in lb is currently set to a. link_2#. (It was set to this value automatically at the time some other procedure in process 2 called < a >.) Now, if and when process 2 is interrupted so that process 1 can take over, the software in MULTICS for process switching automatically saves all machine conditions including the contents of all base registers. This data is saved on a push down list known as the "process stack". (The detailed format of this stack need not concern us. The BJ sections of MSPM provide the details.) When ownership of the process is later regained by process 2, the lb←lp pair, among other registers, will have their values properly restored. Thus, execution in < a > is resumed, and any intersegment references proceed as before - via $<$ a. link $>_2$.

One final bit of good news. There is no change required in the linkage segments of either process as a consequence of switching back and forth between processes!

2.5.1 What If We Didn't Have Linkage Segments?

The following paragraphs are offered for those from Missouri who are not yet ready to accept the need for the < a. link> type of switch box. Others can skip over the following arguments as they see fit.

Suppose a shared assembler, < a > were to use direct intersegment addressing (à la Figure 1-12) in referring to the information in data segments. Suppose process 1 is currently employing < a >.

Now if the processor is switched to process 2, what guarantee do we have that data-referencing instructions, possibly established while using process 1, will now be applicable for referencing the corresponding data segments of process 2? As a matter of fact, even if corresponding data segments have the same names, like < dl > in process 1 and < dl > in process 2, is there any way to guarantee that they will have the same segment numbers in their respective descriptor segments? Quite the contrary, since segment numbers are established on a first come first served (as-needed) basis, there is only a small chance for such a coincidence to occur.

Another way to see this is to observe that each process tends to execute a unique sequence of segments, each calling in some particular order on data segments and other procedure segments. Thus the order of the loading of segments as-needed will tend to give rise to different segment numbers for corresponding or shared segments of two different processes.

It would, therefore, seem fairly clear that to avoid a linkage segment implies the need to change within < a > the data referencing instructions each time < a > receives a new "owner". This contradicts our assumption that < a > might remain a pure procedure so that we could reap the benefits of its purity.

O.K., so we agree that < a > must lose its purity. Let's set about trying to pay the price, hoping it won't loom as large as the apparently costly business of maintaining separate linkage segments in each process for each procedure segment, like < a>. A further question immediately arises. Who alters these instructions in < a> for each process switch, and how and when should it be done? It's fairly clear that another procedure, private to each process, would then be needed to properly alter < a>. This auxiliary procedure then serves < a >, much like a prologue in a subroutine that's generated by a present-day MAD or FORTRAN compiler on a computer like the IBM 7094. A MAD prologue, for example, fetches each argument in the subroutine call and uses it to "set" or complete every instruction which involves the corresponding subroutine parameter. Unlike some subroutines, however, execution of this kind of prologue could never be avoided. It would have to be executed for process 1 each time process 1 obtains control of the processor. The cost of storing and repeatedly executing this prologue could be prohibitive if process switching frequency is high. What's worse, there would have to be a prologue executed for every shared procedure of a process and all such prologues would have to be executed each time the process gets hold of the processor!

This is a nightmare which only gets worse the more one dwells on it. To wake up from this bad dream it is only necessary to remember that it started by suggesting that direct intersegment addressing of data segments might be worth a try.

2.5.2 Avoiding the Extra Memory Cycle in an Intersegment Data Reference*

Use of the linkage segment would seem to imply that every data reference (outside the procedures segment) must be indirect and therefore must involve an extra memory cycle. Conceivable, this requirement could prove costly, for example, in executing the innermost loop of a highly repetitive procedure like a matrix multiply or, for example, where the innermost loop develops an inner product of two vectors. When execution efficiency is really needed, special assembly-level coding may reduce or in some cases even eliminate these inner loop indirect memory cycles.

* This section can be skipped during a first reading without loss of continuity.

A coding technique that offers promise and which takes full advantage of the abr pairs of the GE 645 would be something like:

(1) Load into available abr pairs from its pairs the generalized addresses of data variables that appear in the inner loop. (If the data variable is an array variable, then the its pair loaded into the abr pair would represent a base location in preparation for a "march" through one of the subscript ranges of the array variable).

(2) When inside the loop, reference the data, but via the abr pair for the desired data. This is a direct reference in the sense that no extra memory cycle would be needed.

We illustrate with a hand-coded computation of a simple inner product in the segment <t>, which in PL/I might be written as:

```
innerprod = 0;
loop:  do j = 1 to m; innerprod = innerprod + a(j)* b(j); endloop;
```

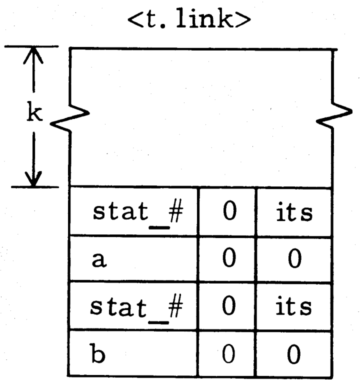
A possible hand-coding (using integer arithmetic) would be:

```
        eapap  <stat_>|[a]      put stat_#| a in ab ← ap, <stat_>is the segment
        eapbp  <stat_>|[b]      put stat_#| b in bb ← bp    containing the data.
        ldxj   1                index reg j ← 1
        stz    innerprod        innerprod ← 0
loop:   ldq    ap| 0, j         direct addressing used in
        mpy    bp| 0, j         formation of a_j x b_j
        add    innerprod
        sto    innerprod
        tix    loop, n, j       some kind of an increment, test, and
                                branch instruction like the IBM 7090
                                TIX.
```

Note that the first two instructions would be assembled in the form:

```
        eapap  lp| k,*
        eapbp  lp| k + 2,*
```

These would refer to locations in <t. link> containing generated fault pairs. These pairs would later be converted to its pairs as shown below.

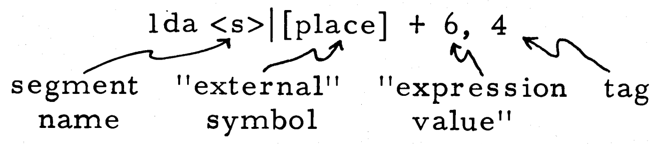


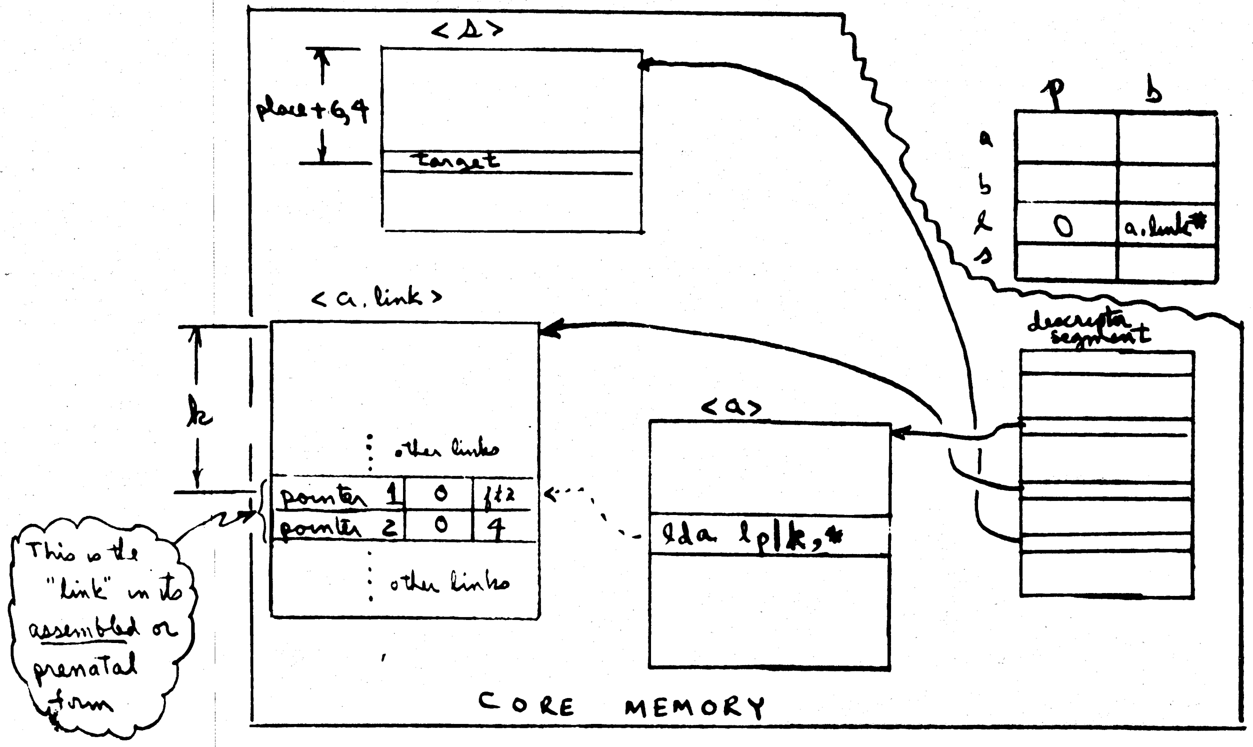
(That is to say, the first time the eapap and eapbp instructions are executed the corresponding fault pairs in <t. link> would be converted to the its pairs shown above.) The ldq and mpy instructions which form a_j x b_j are, as desired, directly addressed rather than indirectly addressed.

2.6 ESTABLISHING LINKS AT EXECUTION TIME

We are now ready to examine many of the remaining details of linking. These are the details which are related to the fact that segments (and linkage segments), are loaded on an as needed basis, making it necessary that the conversion of symbolic intersegment references to numeric references be made at the last possible moment (i.e., at the time the first use of that symbolic reference is made).

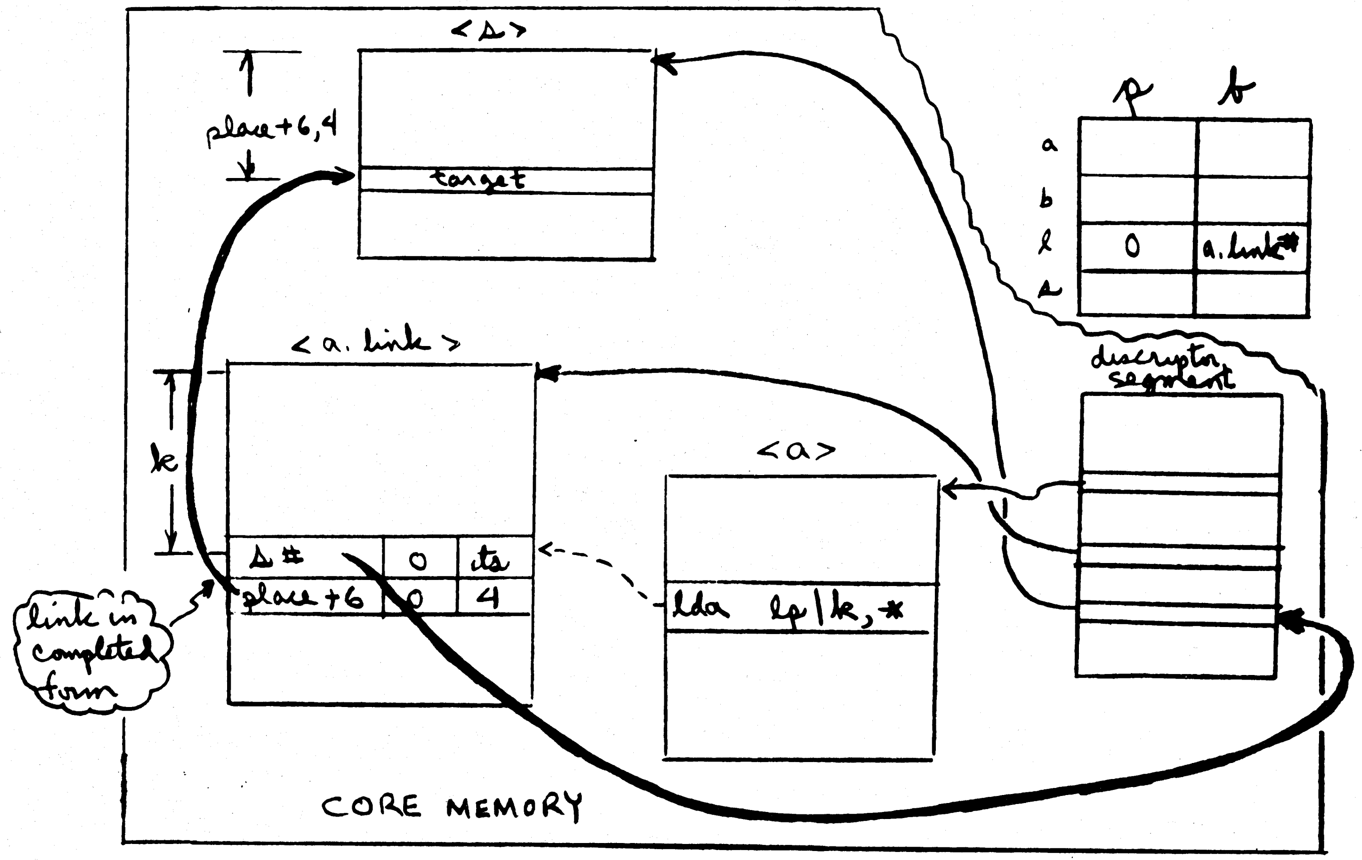
In reviewing what has already been covered on this topic in Section 2.5, we now note that the linking suggested in Figure 2-2(a) and Figure 2-2(b) is an over-simplification. At that time we had not yet developed the concept of the linkage section. Figure 2-5, parts (a) and (b), represent an updating of Figure 2-2, and a short discussion of the new figure will motivate the material in the remainder of this chapter. The illustration in Figure 2-5 is again based on our old faithful instruction:





Symbolic instruction
lda <s>|[place] + 6, 4
is assembled and stored in < a > as the numeric (binary) equivalent of
lda lp| k, *
This instruction points at < a. link >| k, which is the first word of a pair of indirect words. In its initial state, i. e., never-before referenced, it is a link in prenatal form. It contains a special tag in the first word to designate an ft2 fault. It also contains a pair of pointers that lead to symbols < s >, and [place] , and to the value for the remainder of the expression i. e., the value 6 in this case. The fault pair also contains a tag for index register 4.

Figure 2-5(a). Assembling and Storing a Symbolic Instruction



Showing the link in the form of an its pair, as completed by the Linker program. This its pair replaces the ft2 pair. The its pair points at the desired target word.

Figure 2-5(b). A Completed its-Pair Link

As suggested in the preceding section, the assembler translates this instruction into two parts.

a. A single, never-to-be-altered, instruction word whose binary form is equivalent to

   lda lp| k, *

   This instruction is made part of the object code for < a >.

b. A pair of pointers stored in the format of a ft2 word pair. This word pair is placed in < a. link > at a point k locations from word zero of < a. link >.

The symbol k represents a number generated by the assembler. Each time the assembler encounters an instruction with a unique intersegment reference, it generates another fault pair for < a. link >. These pairs are ordered in some fashion relative to a. link #| 0.

The ft2 fault pair contains pointers to all the remaining information embedded within the original symbolic instruction

lda < s >| [place] + 6, 4

2.6.1 <u>The Linker - Phase One</u>

In executing

lda lp| k, *

the GE 645 retrieves the ft2 pair as an indirect word pair found at location a. link#| k. This word pair is shown in Figure 2-5(a). When the special bit pattern denoted by "ft2" is sensed, the GE 645 traps to a special memory location in what is known as the "fault vector". A short program called the "fault intercepter" takes over and saves all machine conditions on a stack. (This is not unlike the action of a trap processor on the IBM 7094.) Next, the fault interceptor causes a special procedure to be invoked known as the "Linker". In the process of giving control to the Linker, the location of the offending ft2 pair is saved. The two pointers that were stored in the ft2 pair are now employed by the Linker to retrieve vital information that is stored in a special list structure known as a <u>link definition</u>. As you might guess, the assembler generates a link definition to go with each ft2 pair. None of the information in a link definition is subject to change. Hence all link definitions are packaged as a single "table". For this we shall use the name "outsymbol table". This table is stored at the tail end of < a > itself, as suggested in Figure 2-6.

The orientation which explains our use of the term "outsymbol table" is this: We associate ourselves with the segment (< a > in this case) that's making an outward reference to another segment (< s > in this case). It's as if we were standing inside < a > and looking outward.) With respect to < a > the symbols in its outsymbol table are "outsiders".

2.6.2. Link Definition Structure (Outsymbol Table)

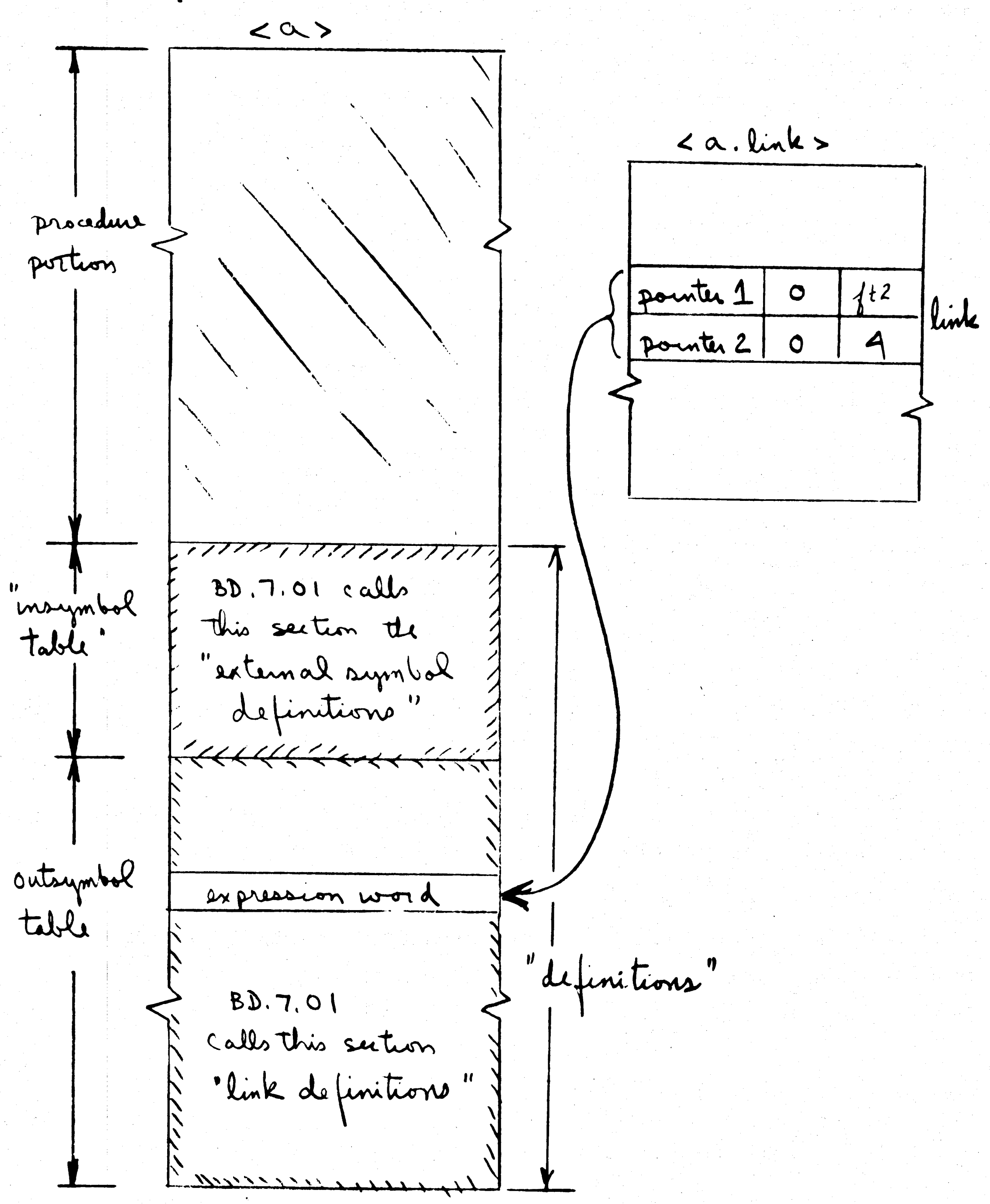
Each definition begins with a header word which in MSPM is referred to as the "expression word". Figure 2-7 shows the particular storage structure of the link definition for our example.* If the use of pointer1 and pointer2 of the ft2 pair allows the Linker to locate the right expression word, it's a simple matter then for the Linker to extract the strings "s" and "place", and the value 6.

There is more to be said about the general structure of a link definition. We have only looked at one type, in particular we've looked at a "type 4" link definition. There are several others, and we'll come back and look at these after we've pursued our current example to its conclusion, i.e., after we have completed the narrative on how the Linker constructs the desired its pair.

s#	0	its
place + 6	0	4

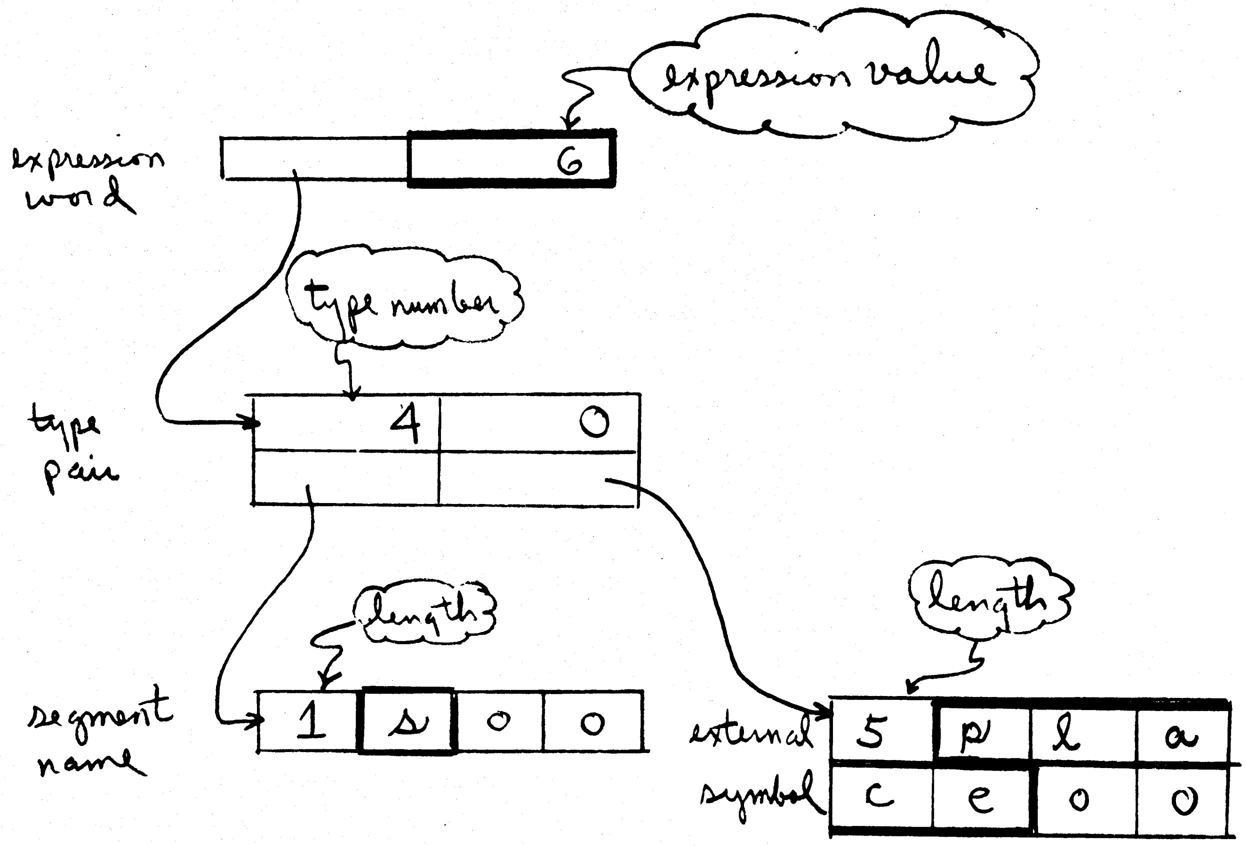
which must replace the ft2 pair.

* Depending on who writes the assembler or compiler that generates the outsymbol table, packaging can be more or less efficient. For the Multics assembler, efficiency is achieved by avoiding duplication of symbol strings. Thus if two or more link definitions refer to the same segment name string or external symbol string, only one copy of the string is kept in the table.



Showing schematic of an ft2 pair pointing at the head of its associated link definition which is located within a portion of < a > known in MSPM as the "definitions" section.

Figure 2-6. Schematic of an ft2 Pair



This structure is for the expression:

<s>|[place] + 6

Note: Character strings are stored nine bits per character (7-bit ASCII, right justified in a 9-bit field), 4 characters per word. Consecutively addressed words are used for strings of length, $\ell \geq 4$ characters. The length, ℓ is coded in the first 9 bits of the first word used for the string. (Maximum length for such strings is seen to be 2^9-1 or 511 characters.) We show the unused portions of a word containing the end of a string as filled with zeros.

Figure 2-7. Data Structure for the Link Definition Stored in <a>

2.6.3 The Linker - Phase Two

The second phase of the Linker's activity is to determine s# from <s>, determine place from [place], and then form and store the desired its pair. The Linker calls on a unit of the superviser known as Segment Management* to help it accomplish this task. We won't bother to spell out, in the description which follows, just who is doing what. We'll just give the Linker "credit" for all of it. Here is how it goes, skipping some of the details.

A search is made of the Known Segment Table (KST) for the purpose of finding s#. If < s > had ever been loaded, (whether or not < s> resides in memory right now), an entry for < s > will exist in the KST. Hence, the corresponding value of s# will be found as a result of this search.

However, when the descriptor field of the descriptor word at s# is examined, it may be found to be of class A, which means the segment is missing. In this event < s > must be reloaded.

If the search of the KST fails to find < s >, it means that no previous reference to < s > had ever been made. Here again, the loading of < s > is required before the Linker can proceed with its appointed task. A search† must then be made for < s > in secondary storage and< s >must be loaded. As < s > is finally loaded, the pair (< s >, s#) is inserted as an entry in the KST. Also, a new descriptor word at s# is added to the descriptor segment. This new descriptor word points to the newly loaded address for < s >.

The linker now knows s#. Its next job is to search a table in < s > for [place] in order to find the corresponding <u>value</u> assigned to it by the assembler or computer that made < s >. When the table entry for [place] is found, the corresponding value for [place] i.e., place, will also be found.

---

* This module may in turn call on other modules known as the "basic file system". More detail is given in Chapter 6 of this Guide.

† The subsystem writer will be interested in learning more about the search strategy which the file system employs here. Details are discussed in BD. 4 and BX. 13. The subsystem writer will find that he can supply the file system any search strategy he wishes, however elaborate, to replace the "standard" search module that is normally provided.

2.6.4 External Symbol Definitions (Insymbol Table)

The table that is to be searched is described in BD 7.01 under the name "external symbol definitions". We are going to rename it the "insymbol table", and we should not confuse this table with the so-called "segment symbol table" (see B.D. 1.00).*

The orientation that explains the term insymbol table is consistent with our "outsymbol" terminology. A segment can not only make outward references but can also receive reference to it, i.e., inward references. Thus, the symbols in the insymbol table for < a > are all those symbols locally defined in < a > which may be referred to by another segment.

The programmer, writing the source code for a given data or procedure segment "marks", via special pseudo ops or other declarations, those symbols whose values he wishes to make known (somehow) to other segments. The assembler or complier then creates an entry in the insymbol table for each of these marked symbols (and their corresponding values).

Each entry has the format shown in Figure 2-8 and the structure for the entire table is pictured in Figure 2-9.

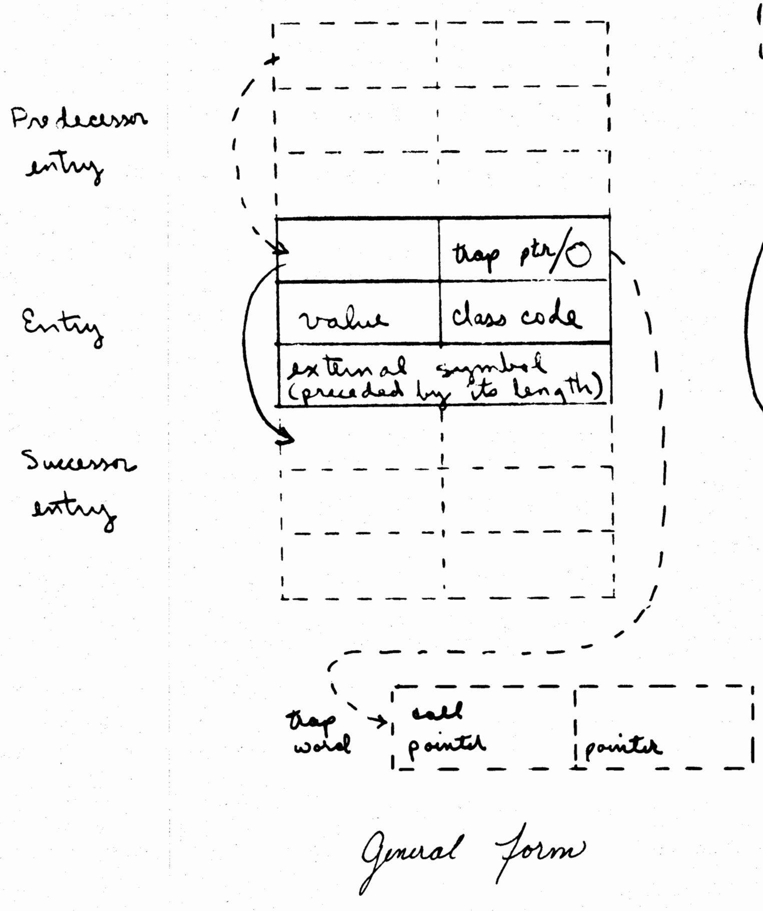
In most instances, the insymbol table, for a segment < s > is invariant under execution, so it can be stored in < s >.† We suggested this idea in Figure 2-6 where we show the insymbol table placed immediately following the executable code of the segment. The insymbol table always precedes the outsymbol table, by convention.

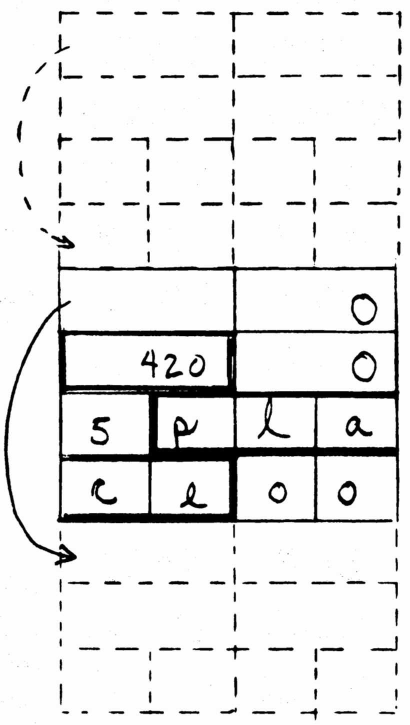
Each symbol entered in the insymbol table for < seg > is assigned a class code. The class code is used mainly to designate to which segment — i.e., < seg>, < seg. link> or < seg. symbol> — the particular symbol and its value refer.

---

* The segment symbol table for a prodecure segment is a much more elaborate table. It associates with each symbol lists of attributes and other information valuable for debugging aids, data directed I/O, etc. --in addition to the address of the symbol. This table is actually stored in a separate but related segment. Thus, a procedure < x > has associated with it really two segments. One of these is < x. link > and the other is <x. symbol >. Together these three segments form a "group". When the linker, operating for a given process, decides <x> is needed in memory, copies of < x. link > and < x. symbol > are then automatically loaded and their corresponding segment numbers are placed in the KST.

† A definition which holds a trap pointer, called a trap-before-definition pointer is an exception. These definitions are employed in special situations where a trap to a special procedure is desired at the time the Linker searches for this symbol the first time. Upon return from a trap procedure the Linker will reset this pointer and hence an in-symbol table containing such definitions would be stored in the linkage segment. Chapter 5 illustrates the use of the trap-before-definition in connection with establishing default handlers for system-defined conditions.





Specific example.
Symbol is "place."
Its value is 420

In the specific example, the symbol is "place" (of length 5). Its <u>value</u> is imagined here to be 420. Assuming this entry is in the insymbol table of <s>, we see that the sought-after value of [place], which we have been denoting symbolically as <u>place</u>, is actually 420. The class code is 0 in this example, which means that place (=420) is an offset within <s> (rather than within <s. link> or <s. symbol>).

Figure 2-8. General Form of an Insymbol Table

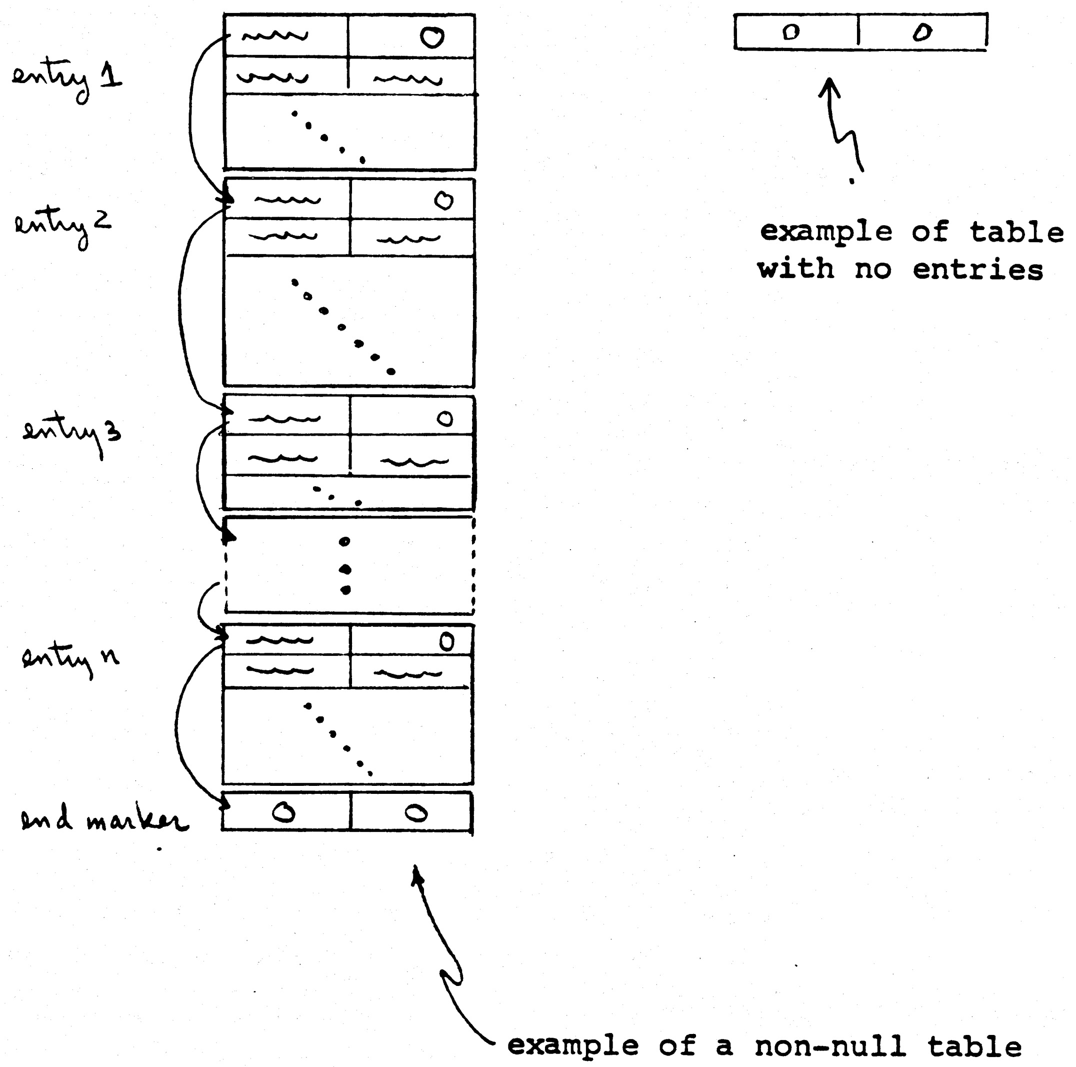


Figure 2-9. Format of the Insymbol Table.

Class Code	Meaning	Remark
0	value is interpreted as an offset from top of <seg>	
1	value is interpreted as an offset from top of <seg. link>	procedure entry point
2	value is interpreted as an offset from top of <seg. symbol>	

We'll have more to say about the class code when we speak about entry-point references in Section 2. 9. (Another use of the class code is discussed in connection a Topic referred to as signalling in Chapter 5.)

2. 6. 5 The Linker — Phase Three

When the value of $\left[\text{place}\right]$ has been obtained, the Linker is now close to finishing its job by completing and storing the link and resuming execution. It has determined s# and place. Now it generates the its pair you see in Figure 2-5(b) and stores this in place of the fault pair. (Remember the location of the fault pair was stored as part of the trapping action.) Now the Linker returns to the "Fault Interceptor" which called it. The Fault Interceptor restores all machine conditions which existed at the time of the fault (popped from a special stack), but now corrected to reflect a stored its pair, and the GE 645 processor then resumes the execution of the lda instruction which cased the fault.

2.7 MORE ON THE STRUCTURE OF LINK DEFINITIONS

There are five types of symbolic intersegment references one can make in assembly language source code. For each type there is a different structure generated for the link definitions and a different type of link is generated. We have already looked at one of these types. It's called a "type 4" reference. Types 2, 3, and 4 are listed and illustrated in Figure 2-10.*

The link ultimately generated in a type 2 reference is necessarily an itb pair, while those generated for other types are its pairs.

* Types 1 and 5 are special purpose variations of types 3 and 4, reserved for self references, i.e., references to the executing procedure, to its linkage segment, or to its symbol segment. More details may be found in BD. 7. 01.

Type No.	Source Code Examples	Syntactical Form Of Intersegment Reference	Ultimate Form Of Generated Link			
2	bp| [blue] + χ - 6, 7*	base| [ext] + exp, m	base x 2^{15}	0	itb	general
	means segment tag = 2		exp + ext	0	m	form
	We shall assume χ = 2012 as set by the assembler		2 x 2^{15}	0	itb	specific
			blue + 2006	0	7*	example
3	<weight>| mid + 50	<seg>| exp, m	seg#	0	its	general
	We shall assume the assembler makes mid = 3422		exp	0	m	form
			weight#	0	its	specific
			3472	0	0	example
4	<s>| [place], 3	<seg>| [ext] + exp, m	seg#	0	its	general
			ext + exp	0	m	form
			s#	0	its	example
			place	0	3	1
	<s>| [place] + key - 6		s#	0	its	example
	We shall assume the assembler defines key = 100		place + 94	0	0	3

Key:	base	means base register number
	m	means modifier
	exp	means expression
	ext	means external symbol

Note that either exp, or m, or both may be omitted without altering the type number.

Figure 2-10. External References—Types 2, 3, and 4

Figure 2-11 shows the storage structure for the link definitions that go with each of the three types of reference.

It's true we've already illustrated the storage structure for a type 4 link definition. This was given in Figure 2-7. Nevertheless, we give a second example in Figure 2-11(c) to illustrate the important "trap-before-link feature" of the Multics linkage mechanism.

2.8 THE TRAP-BEFORE-LINK FEATURE

2.8.1 Why Have It?

The trap feature has been incorporated in Multics for benefit of some subsystems writers who must have the object programs that are produced by their subsystem, like PL/I or FORTRAN, automatically perform certain special storage management tasks at the time certain segments are first referred to. For example, by inserting this trap feature in certain link definitions the object program can create or grow segments as allocation for variables which, in PL/I terminology, are referred to as static-storage variables. For variables of either static-storage or automatic storage the trap procedure can also assign initial values for those variables that are declared to have the initial attribute, i.e., at the time the space for such variables is being created.* In other instances, it is conceivable that the

* At the time the data space is allocated, for a static variable an entry must be created and added to the insymbol table for the data segment.

This table entry will provide the value of the symbol, i.e., a pointer to the allocated slot in the data segment so that other links to this same location can be completed.

These other links may be required for references to the same data location, either from the same procedure segment that makes the initial reference, or, if the variable is static external (like COMMON or PROGRAM COMMON in FORTRAN or MAD), from other procedure segments.

Unlike insymbol tables for procedure segments, which are kept in the procedure itself, the insymbol table for a data segment is normally kept in the linkage segment associated with the data segment. In this way the data segment can grow as repeated allocations are made and at the same time the corresponding insymbol table can also grow independently.

Adding an insymbol table entry, along with that of extending the length of a data segment and possibly assigning initial values --- all these services must be "standard apparatus" for a compiler in Multics. For more details see BP.4.00 and especially BP.4.01. Also see BY.13.

trap procedure could be used to monitor (gather statistics) on segment usage for a set of object programs. It's likely that the subsystem writer will dream up many applications for this feature (but he can also leave it alone). There will be no penalty paid for having the feature available and not using it.

2.8.2 What Is It and How Does It Work?

Suppose a user wishes to gain control (i.e., intercede) just before the link for some intersegment reference is completed. For example, let the instruction be

ℓda <s>| [place], 3

Let the procedure that is to be executed before establishing the link to <s>| [place], 3 be

<proced>| [begin]

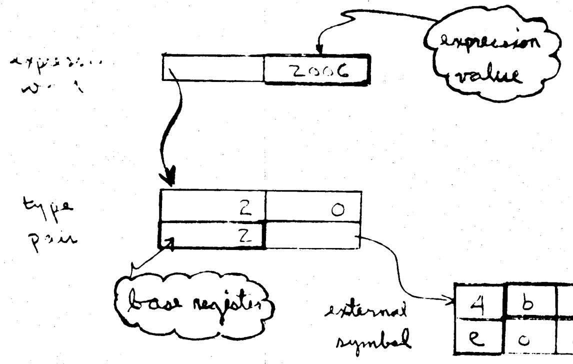
Also, let the argument list for this procedure be located at <param>| 0.

When the Linker is busy searching through the link definition for the reference

<s>| [place], 3

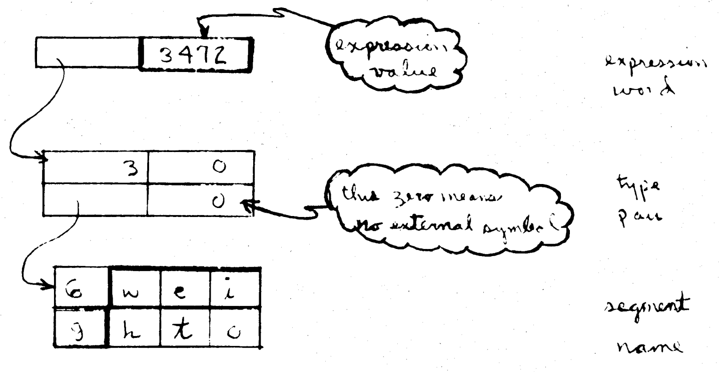
as shown in Figure 2-11(c), it will discover a non-zero value in the right half of the type pair. A non-zero value will be interpreted as a trap pointer and this will immediately cause the Linker to digress for the purpose of executing the trap feature. By working its way over to the links that contain the name of the procedure and the information about the argument list, the Linker is able to generate and execute the call that executes the desired "trap" procedure. More details are given in BD 7.01. Suffice to say, that upon return from the trap procedure the Linker will again have control, so it can now complete the link that it started out to build. In this case it is, of course,

s#	0	its
place	0	3



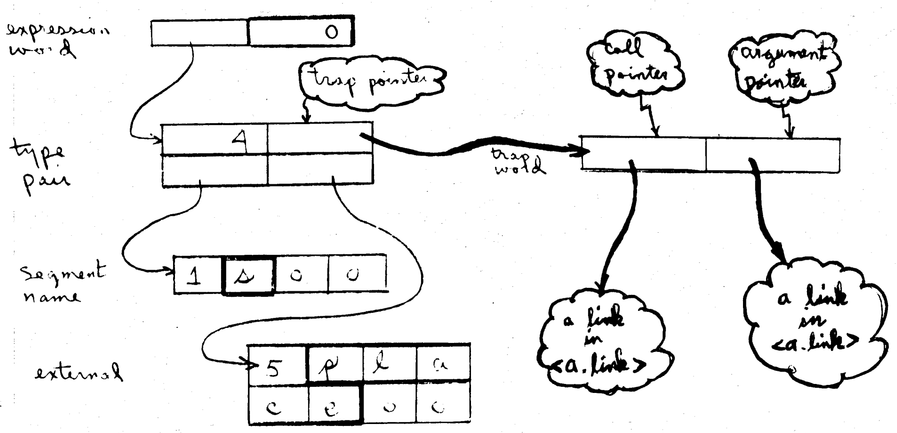
(a) Type <u>2</u> reference:

bp|[blue] + 2006, 7*
(See Fig. 2-11(a))



(b) Type <u>3</u> reference:

<weight>| 3472



(c) Type <u>4</u> reference with the trap feature invoked.

<s>|[place], 3

Normally, the right half of the first word of the "type pair" is zero. In this case, however, it contains a so-called <u>trap pointer</u> which leads to a pair of other pointers used by the linker to generate a call on an arbitrary, user-specified procedure.

Figure 2-11. Structure of Link Definitions

2.9 TRANSFER TO A PROCEDURE ENTRY POINT

Let's go back to Figure 2-1 and consider the linking process that's involved in executing the transfer instruction: †

```
tra <t>| [entry2]
```

In transfering control from <a> to a point within another procedure segment, <t>, the linkage is necessarily more complex because, before beginning to execute in <t>, we want to load the lb ← lp base pair so that it points at < t. link > instead of < a. link >. No special GE 645 hardware is available to accomplish the change in lb ← lp base register values automatically. It can only be done by having the assembler convert the tra instruction into a short sequence of several instructions.

We will see that the resulting code will force the Linker to do double duty. The following is a brief sketch of the steps involved:

1. In <a>: execute an instruction of the form

   ```
   tra  lp| k, *
   ```

   which is a transfer through an indirect word pair found in <a. link>.

2. In <a. link>: the indirect word pair, pointed at by

   ```
   lp| k,
   ```

   will be the link:

t. link#	0	its
number1	0	0

   When this link has been completed after invoking the linker, we now have a transfer from < a >, through < a. link >, to to < t. link >| [number1] .

† Such an instruction would normally appear in a program as a result of issuing any procedure call. In assembly language, for instance, the use of the "call" macro generates a stereotyped (but critically important) sequence of instructions called the CALL sequence. See BD. 7. 02for details.

3. In < t. link>: a pair of instructions and a related ft2 indirect word pair are provided. The instructions are located at < t. link>|[number 1] and < t. link>|[number 1] + 1. The first of these is an <u>eaplp</u> instruction which effectively causes lb ← lp pair to be loaded as follows:

   lb   t. link#
   lp   offset of beginning of the linkage block in < t. link>.

   The second instruction is an indirect transfer (through the word pair that is also within < t. link>) to < t>|[entry].

   The indirect transfer is accomplished by using an ft2 fault pair. The Linker must again be invoked to convert this ft2 pair to a proper link. This time the completed link will look like:

t#	0	its
entry2	0	0

   And this last step completes the linking for the intersegment transfer.

In order to complete this second link, the Linker first searches a link definition that is found within the outsymbol table of < t>. Here it discovers that the segment being referred to is < t> itself, i. e. , a <u>self</u> <u>reference</u>. Moreover, since the link definition shows a type 3 reference, there is no external symbol whose value needs to be determined. The value has been preset by the assembler and placed in the right half of the expression word. So, the insymbol table for < t> need not be searched. A self reference is discovered by the Linker whenever that pointer in a link definition, which ordinarily points at the character string for the segment name, turns out to be zero.

<u>In summary</u>, the work involved in an intersegment transfer is seen to be:

1. Execute an indirect transfer from < a >, through < a. link >, to < t. link>. In doing so, invoke the Linker(first time only, of course).
2. Execute an eaplp instruction in < t. link > to set lb ← lp for t. link#.
3. Execute an indirect transfer from < t. link >, through < t. link >, to < t >. In doing so, invoke the Linker again (first time only, of course).

Figure 2-12 displays core memory representations for parts of <a>, <a.link>, <t>, and <t.link> during the course of establishing the two links.

You might already be wondering why are we transferring first to <t.link>. Why not go to <t> directly? The answers are these:

(a) Loading the $\ell b \leftarrow \ell p$ pair with t.link# takes only one (eaplp) instruction when executed from <t.link>. To accomplish the same task from <t> is practically impossible. The footnote explains why.*

(b) We are going to need <t.link> in memory anyhow, in order to execute in <t>.

Next you may have wondered by what magic was it possible to establish the first link as

t.link#	0	its
number1	0	0

instead of as

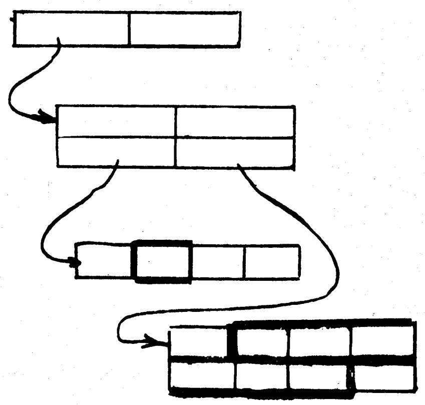
t#		its
number1	0	0

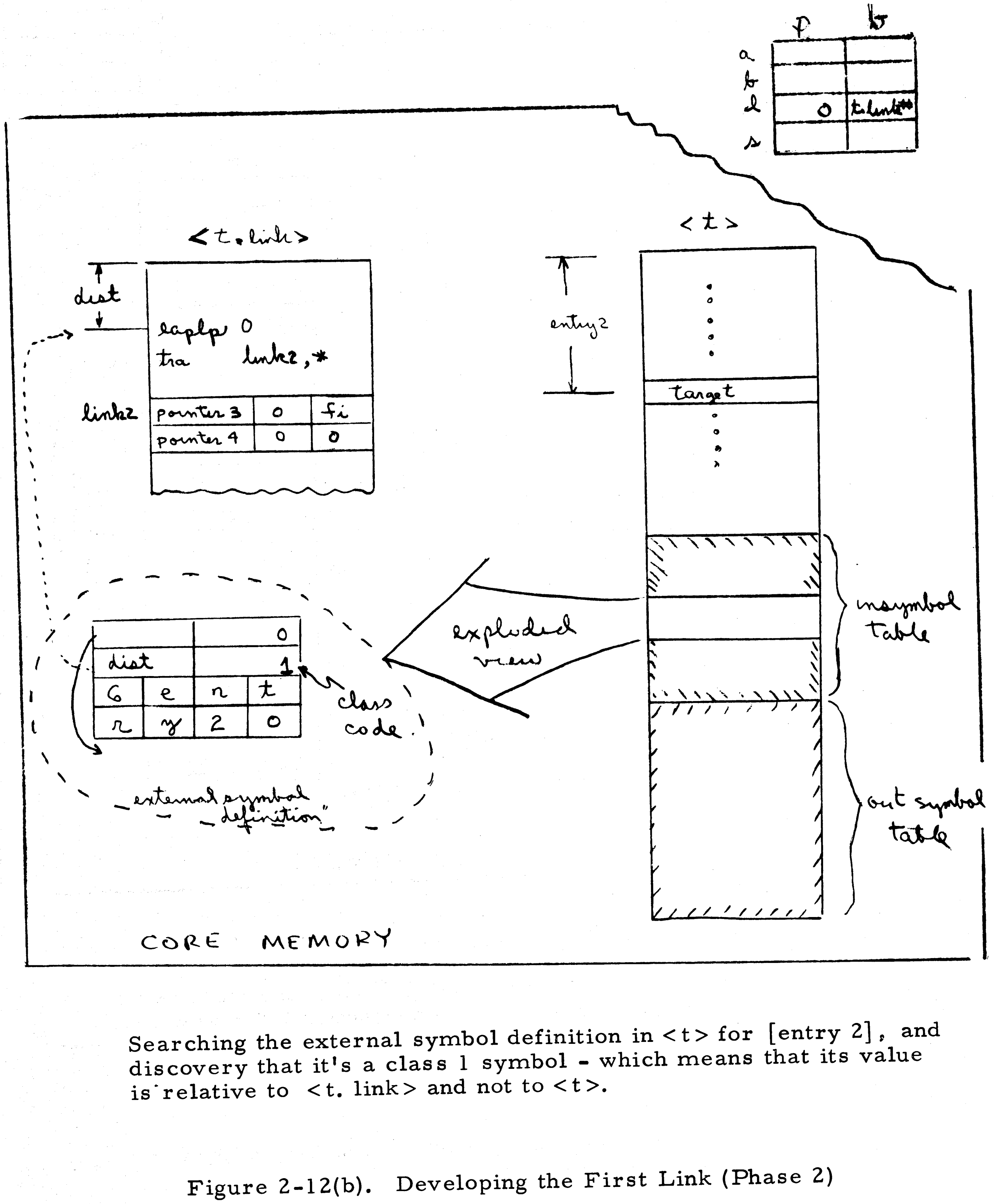
* If we transferred to <t> directly, how would <t> know the value of t.link# to use in loading the lb ← lp? One approach might be to have <t> call on a supervisory procedure to determine t.link# from a search of the KST. This would involve executing a large number of instructions. But wait a minute! How can <t> call on any procedure to help to find t.link#? Calling on another procedure implies that an instruction of the form

op code $\ell p|$ k,*

can be executed, where $\ell b \leftarrow \ell p$ contains t.link#. But, of course, if we had t.link# loaded into $\ell b \leftarrow \ell p$ we wouldn't need to call on a supervisory procedure for help. We see, therefore, that transferring to <t> before loading $\ell b \leftarrow \ell p$ with t.link# would have the effect of hopelessly isolating <t> from any communication with the rest of the system. In other words a transfer to <t> in this way would result in a dead end with no way to return except possible to the segment that called it. Clearly this could not be a generally useful approach.

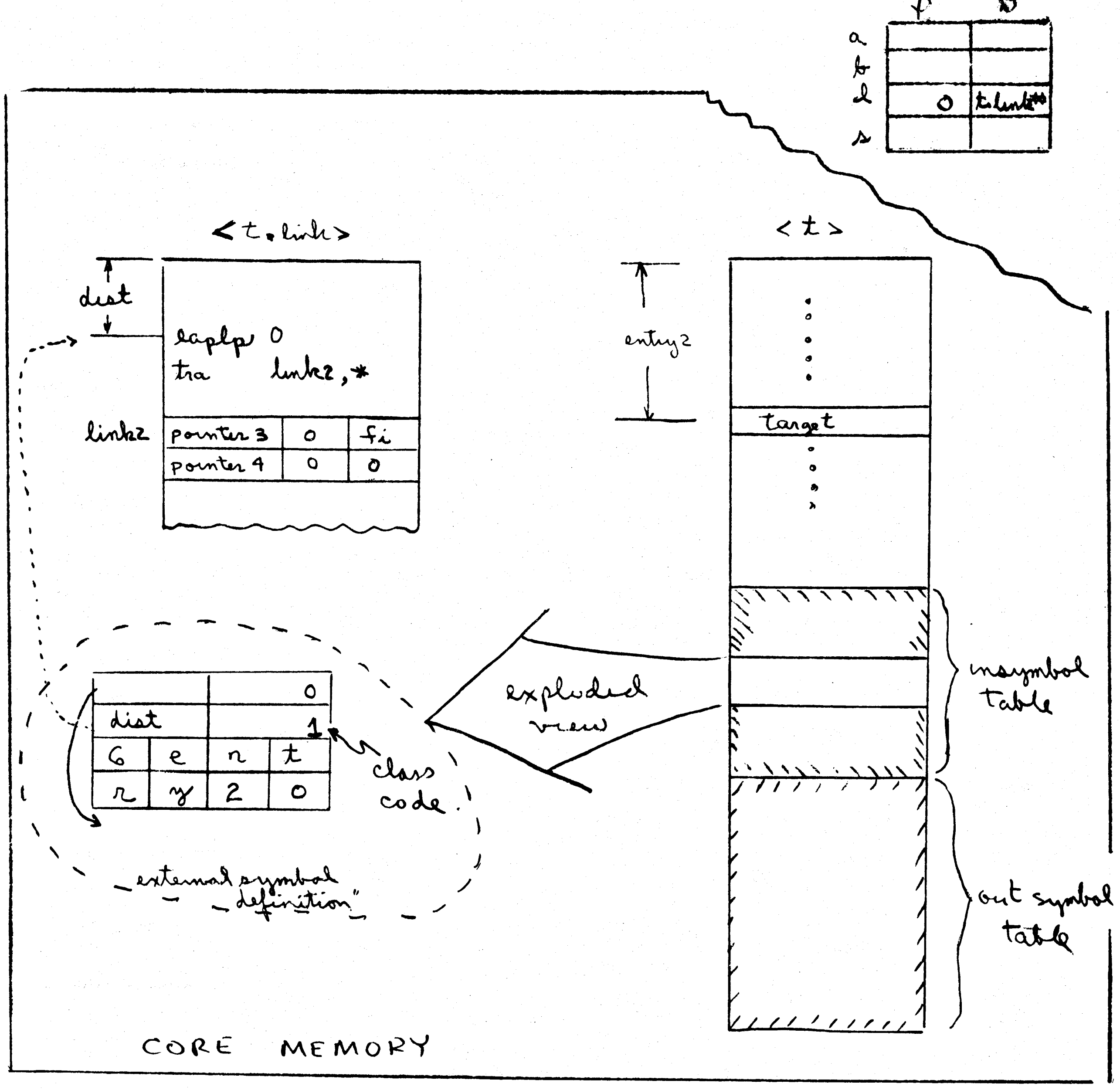
A second approach would be to design Multics so that there's always a predetermined relationship between t# and t.link#, such that instructions executed in <t> could evaluate t.link# and then load it into $\ell p \leftarrow \ell p$. It turns out that such a plan would impose restrictive conventions in the numbering of segments which are incompatible with the operational flexibility needed for segment management in Multics.





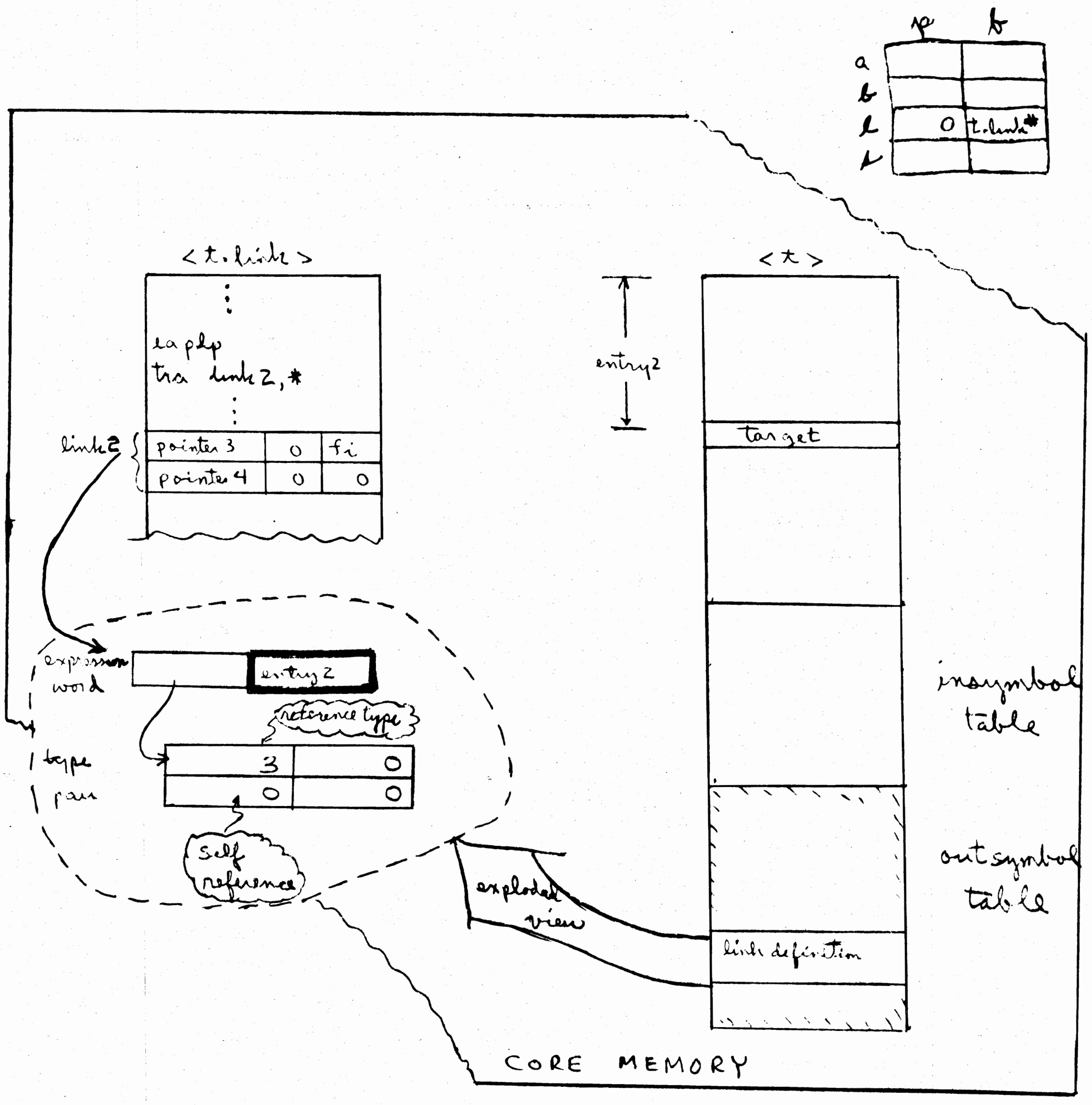
Searching the external symbol definition in <t> for [entry 2], and discovery that it's a class 1 symbol - which means that its value is relative to <t. link> and not to <t>.

Figure 2-12(b). Developing the First Link (Phase 2)



Searching the external symbol definition in <t> for [entry 2], and discovery that it's a class 1 symbol - which means that its value is relative to <t. link> and not to <t>.

Figure 2-12(b). Developing the First Link (Phase 2)



Searching the link definition in <t>. Here the loader Linker discovers that it's a type 3 reference, and moreover it's a self reference. Consequently the linker accepts the value of the expression (stored in the right half of the expression word and shown as [entry2]) as the effective internal address within <t>. That is to say, no phase 2 is needed. The second link can now be generated as shown in Figure 2-13 (d).

Figure 2-12 (c). Developing the Second Link (Phase 1)

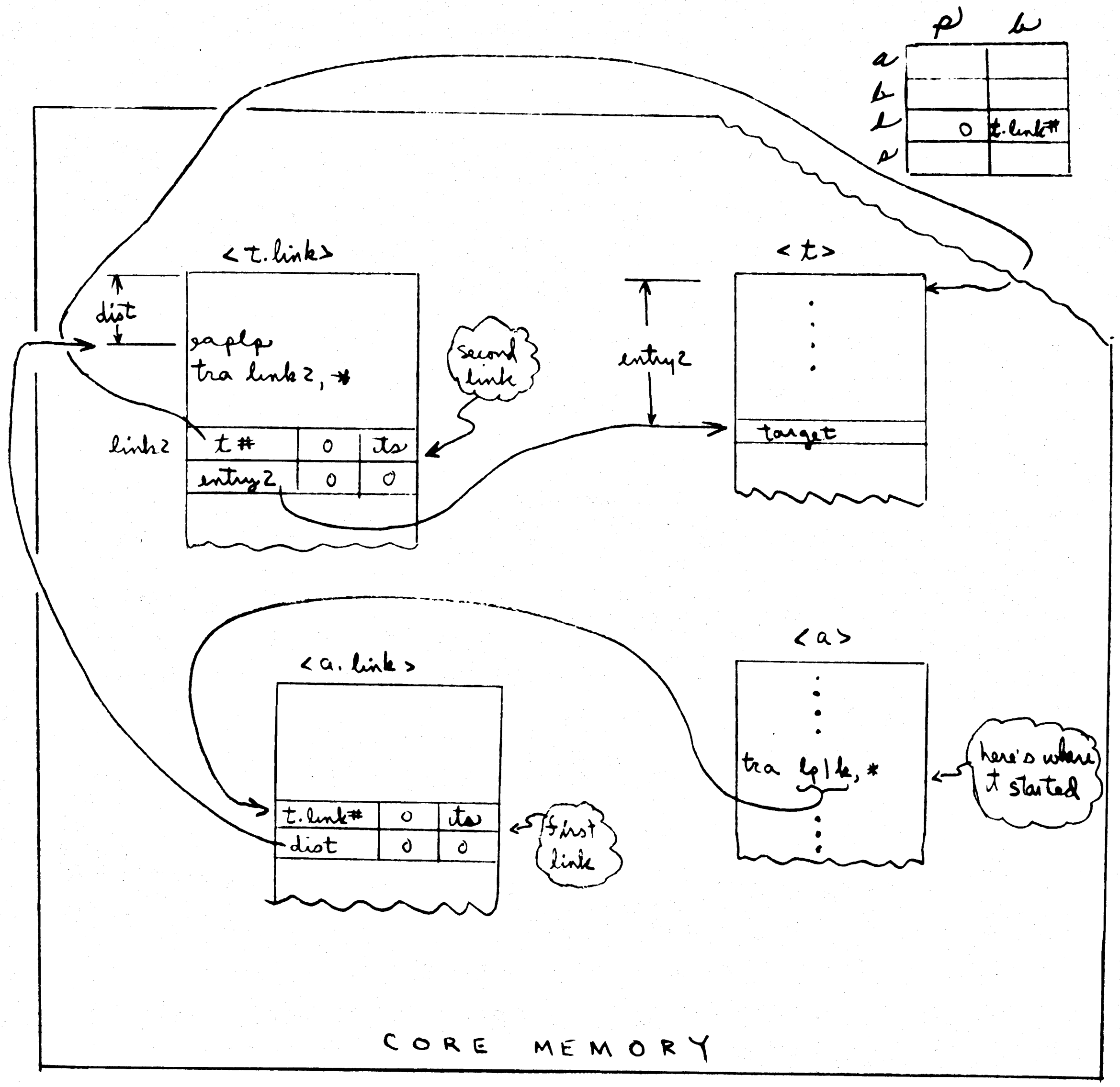


Figure 2-12 (d). A Composite of Both Links completed for the Intersegment Transfer from <a> to <t>

After all, the original instruction read

tra <t>|[entry2]

and not

tra <t. link>|[entry2].

To answer this we call your attention to Figure 2-12 (b), where you see that, in searching the insymbol of <t> for [entry2], we find it to be a class 1 symbol. You will recall from Section 2. 6. 4, that a class 1 symbol is interpreted to mean that the proper segment number to use with the value of [entry2] is t. link# rather than t#.

2. 10 FORMAT OF THE LINKAGE SEGMENT

In preceding sections we have implied that the Linker is able to use the pair of pointers which it finds in the ft2 fault pair to locate the expression word in the link definition. For example in Figure 2-12 (a) we showed the fault pair

pointer1	0	ft2
pointer2	0	0

in <a. link> and we suggested that the Linker employs pointer1 and pointer2 to locate the proper link definition in <a>. To explain how these pointers are used it's helpful to digress for a look at the underlying format given to each linkage segment.

A design objective of Multics is that there be a capability of easily combining or binding together into one segment the separate linkage segments of two or more data or procedure segments. Suffice to say that, in the interests of efficiency, certain groups of supervisory procedures have their linkage information bound into a single segment. To make it convenient to bind such information each linkage segment is structured so that it can become one block of a two-way linked list of blocks. Thus, every linkage segment when first generated, consists of an eight-word block header, followed by a "body", made up of entries and links belonging to this block and/or insymbol table entries (for data segments).

The block header contains three pairs of words (which are pointers) plus the length of the block. The eighth word is unused. The second and third word pairs are preset to zero and remain zero as long as the block is a "loner"; i. e., is not two-way linked to any other block. (The second and third word pairs of each block header are provided so that they can be made into its pairs and serve as "forward" and "backward" pointers to other block headers in a list of such blocks.)

The first word pair of the block header is used in one of two ways depending on the nature of the linkage segment.

1. A linkage segment for a procedure segment uses this first word pair as an its pair that points to the beginning of the (insymbol and outsymbol) definitions within the procedure segment.

2. A linkage segment for a data segment uses only the first word of the first word pair. It's used as a single indirect word pointer to the beginning of the data segment's insymbol table definitions that are stored within the linkage segment itself.

Figure 2-13 illustrates the important connections for a single-block linkage segment referring to the procedure segment <a>. We note from this figure the following points:

(1) Pointer1 of the ft2 pair is "self-relative" to the top of the block header where we see an its pair that points to

a#| defa

This is the beginning of the definitions section within <a>.

(2) The assembler which generates <a.link> cannot, of course, know a# but it can and does know defa, so the as-generated (initial) condition of the first its pair in the block header is really set as:

0	0	its
defa	0	0

When <a> is made active (i. e., gets a segment number assigned to it), the Linker will be invoked to complete this its pair as you see it in Figure 2-13.

For a more specific example, suppose we consider the link called "link2" that's shown in Figure 2-12 (c). If this link is located 50 words from the top of <t.link>, then the value generated for pointer3 will be -50.

(See Figure 2-14 (a).) The pointer in the first word of an ft2 pair is called the "head pointer".* It is self-relative, pointing to the first word of the block header. The its pair located there points in turn to the top of the definitions section in <t>. It's called the "definitions pointer".*

Suppose the expression word of the desired link definition is 80 words down from the top of the definitions section in <t>. Then the value generated for the second pointer of the ft2 pair will be 80. This pointer is called the "expression pointer".*

Since the linker is handed the core location of the ft2 pair, it can determine the location of the definitions pointer by the relation:

loc. of definitions pointer = location of ft2 pair + head pointer.

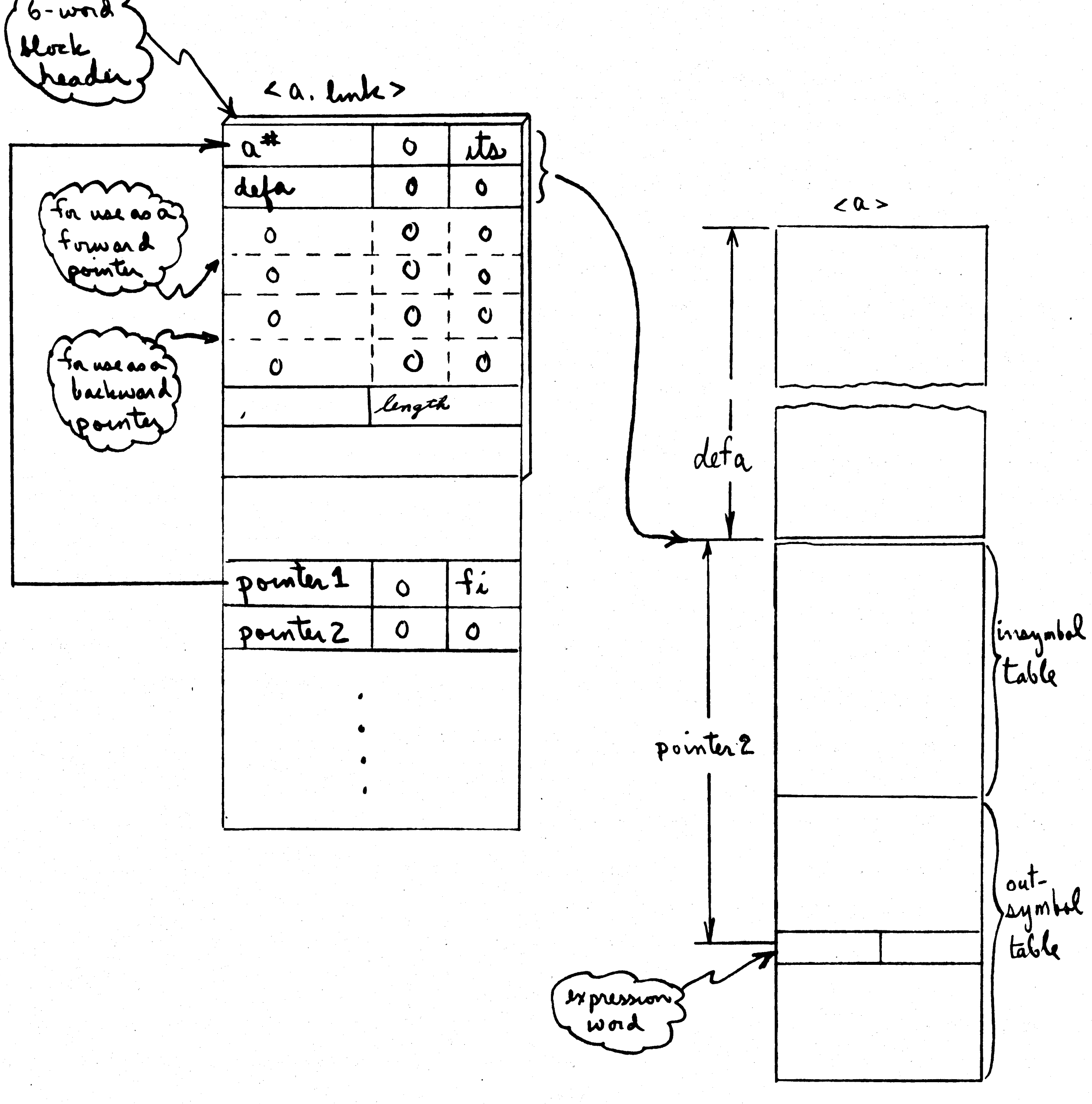
The pointer found here, together with the expression pointer, is then used to construct the location of the expression word which, in this example, is:

t#| 250+ 80.

Figure 2-14 (a) shows the details of this example. Figure 2-14 (b) is a slight generalization using the terminology you would see in the reference document BD. 7. 01.

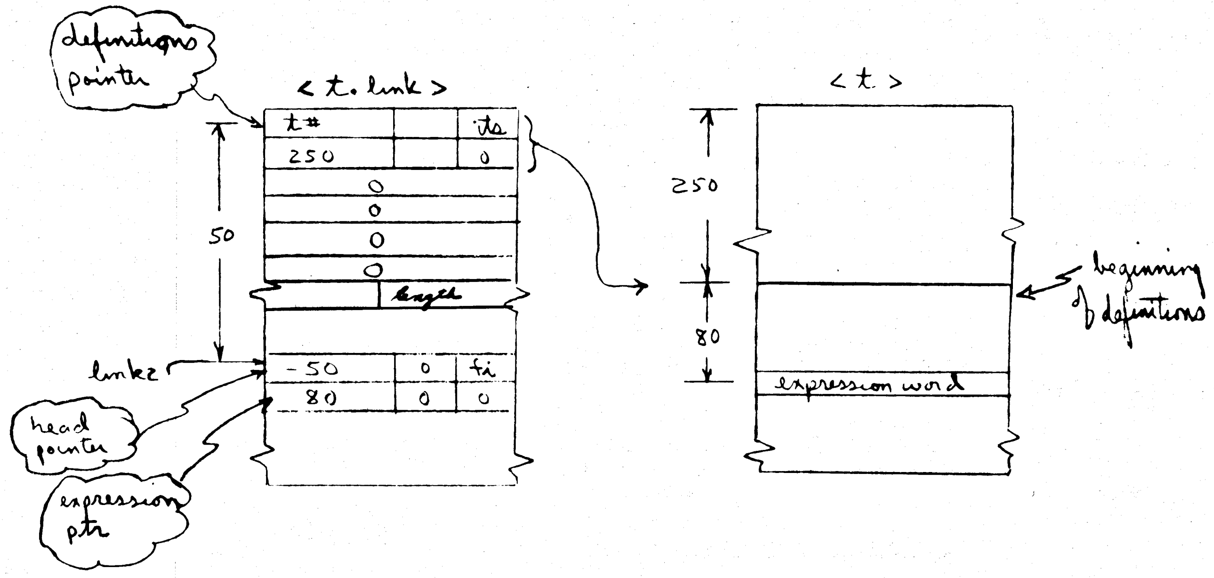
Figure 2-15 shows the appearance of a linkage segment for a data segment, <d>, containing only insymbol table definitions. The first word of the header provides an indirect word whose address is that of the body of this block. Figure 2-16 shows the appearance of a linkage segment for an impure procedure <impa>. We imagine that this linkage segment has two blocks. The first block would consist primarily of the information generated by the assembler of <impa>. It is χ words in length. The second block linked to it could contain new insymbol table definitions that refer to locations named during execution. The user can add a block of such definitions using the procedure calls described in BY. 13. The first word of the second block would contain the effective internal address $\chi + 8$ which locates the definitions in this block.

* Terminology used in BD. 7. 01.

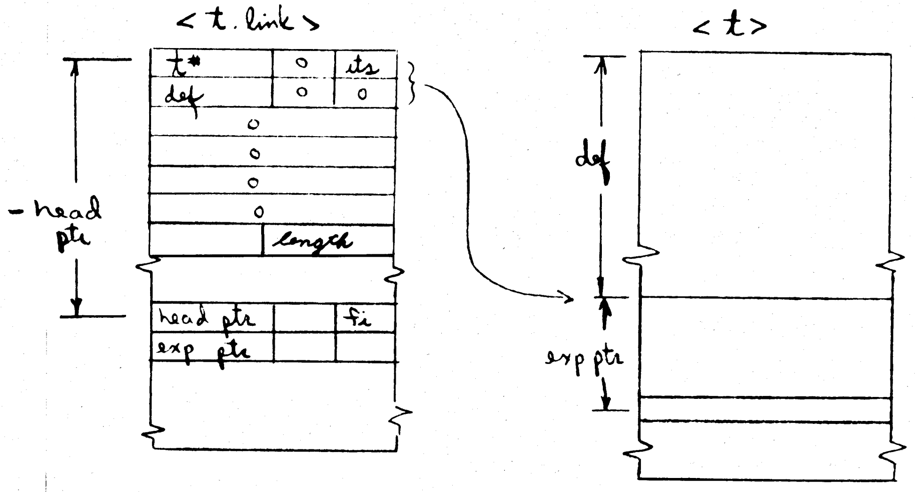


This illustrates the pointer system from an <a. link> ft2 pair to the expression word in < a >

Figure 2-13. Single-Block Linkage Segment

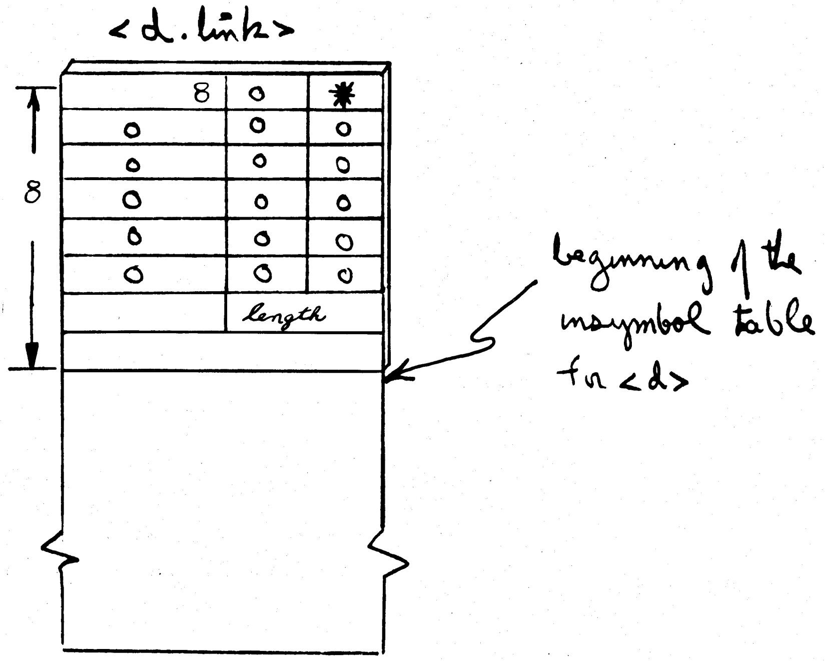


(a) Details of the Example



(b) Generalization of the Example

Figure 2-14. Single-Block Linkage Segment Link2



The first word of < d. link> is an intrasegment pointer (single indirect word) to the beginning of the insymbol table entries for the data segment < d> .

Figure 2-15. Intrasegment Pointer in <d. link>

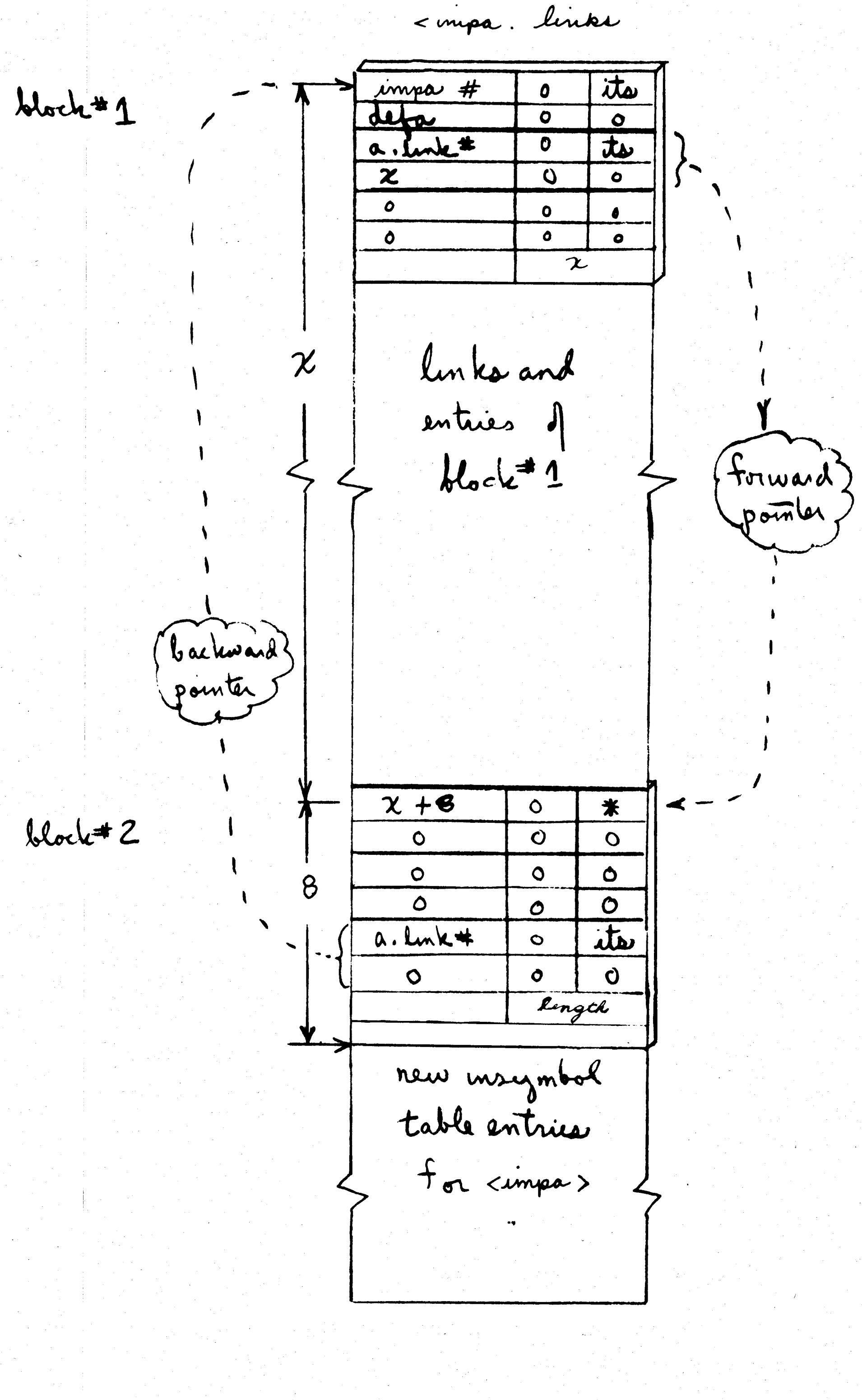


Figure 2-16. Showing a Second Block Added to <impa. link>

2.11 SELF RELATIVE ADDRESSING USED FOR THE ENTRY INSTRUCTIONS OF LINKAGE BLOCKS

Since linkage blocks can be added to form a chain of such blocks in one segment, there is a need to observe one oversimplification which we made in an earlier discussion that can now be removed. The point we are about to make is a subtle one and need not be considered on first reading.

Note the pair of instructions at <t. link>|dist in Figure 2-12(d), and observe that in using these instructions we have assumed that the linkage block shown there begins at word zero of <t. link>. Suppose for reasons of efficiency we would like to bind several procedure segments (and their corresponding linkage segments (see BD. 2). It might happen that the <t. link> block shown in Figure 2-12(d) would now be appended to the end of another block in a newly-formed composite linkage segment. Now the pair of instructions:

```
dist:  eaplp  0
       tra    link2,*
```

no longer do their jobs. It's necessary in practice (and in fact it's a Multics standard) to use the following types of instructions to achieve self-relative addressing:

```
dist:  eaplp  -*,ic
       tra    link2-*,ic*
```

Here the modifier "ic" means add the current contents of the instruction counter to the address value designated in the address field of the instruction.

Thus

```
dist:  eaplp  -*,ic
```

means establish in lb←lp the values:

$\mathrm{lb} \leftarrow$ segment number of executing procedure

$\mathrm{lp} \leftarrow \underbrace{\text{(ic) - dist}}$

this is the address of word zero
of the containing linkage block for the
eaplp instruction regardless of whether
the block itself begins at word zero of
the segment in which it's embedded.

When referring to these "entry" instructions in the future, as in Chapter 3, we shall illustrate with the standard forms introduced in this section, rather than the simplified versions shown previously.