Published: 05/21/68

Identification

The Attach Table and Attach Table Maintainer
D. A. Levinson

Purpose

The purpose of this document is to describe the contents of the Attach Table and the calls to the Attach Table Maintainer by which items of Attach Table entries are set, altered and referenced, and by which entries are created and deleted. Table 1 is the EPL declaration of the Attach Table entry. The items are discussed more fully in the following.

Structure of the Attach Table

Figure 1 displays the Attach Table (AT) as a per-group segment with a local (per-process) extension (labeled LAT for local attach table in Figure 1) for each process requiring one. (The LAT's are discussed below. In Figure 1 only Process 2 and Process 4 are shown with LAT's.) As shown, an AT Template resides in file system storage and is known by name to the ATM. The AT Template is an initialized but otherwise empty Attach Table Segment. On first reference to the ATM in a group, the Attach Table Maintainer calls the SMM (Segment Management Module) to create the per-group AT segment and copies the AT Template into it. On each reference to the ATM in a new process, the ATM calls the SMM to make the existing per-group AT segment known to the new process. In contrast with the AT which is a segment shared by a user-group, the LAT is an unshared segment of the process in whose behalf it was created. Once, they are obtained from the SMM, pointers to the AT and LAT reside in the ATM's per-process (internal static) storage. Finally, the AT Template "pointer" is simply a literal in the ATM procedure.

Figure 2 shows the Attach Table Structure as a three-level threaded list. The structure is intended to facilitate searching and avoid varying-length entries.

The ioname level entries contain the process-invariant items of the AT. The key for searching this level is of course the ioname itself.

The process level entries of the AT contain the process-dependent items of the ioname entries. To explain further, some of the items in an AT entry are pointers. Pointers to the same data in different processes have different forms. Specifically, the segment number (but not the offset) is different in different processes. This is handled by maintaining the offset of a pointer as a process invariant item at the ioname level of the entry, and the segment number as a process-dependent item at the process level. A process level subentry appears under an ioname

for each process in the group which process has referenced the ioname at least once.

The Local Extension of the Attach Table

Ionames may be attach locally, that is, such that the ioname is known only to the attaching process. It is important to note that if an ioname "alpha" is attached locally, then during the term of its attachment the given process cannot operate on a simultaneously attached global alpha. A reference to an ioname "alpha" is taken as a reference to the local ioname "alpha" if one exists. This is built into the ATM's search algorithm. It is also worth noting that this does not restrict a process locally attached to "alpha" from attaching a global alpha, but it does prevent it from referencing global alpha.

When an ioname is attached locally by a process, the AT entry for that ioname appears in an AT extension for the attaching process (the LAT of Figure 1). The LAT is created in a given process only if a localattach call is issued in that process. If an LAT does exist, when an ioname is referenced, the ATM always searches it first, and only if the search of the extension fails does it continue the search in the global (per-group) AT.

The Items of an Attach Table Entry

All of the items of an Attach Table entry are briefly discussed in this paragraph. (Complete discussions of items not directly connected with the switching complex are given in the papers to whose matter these items are primarily relevant.) For example, ioname1, type, and ioname12, are discussed in detail in MSPM Section BF.1.01, Attachment and Detachment of Input/Output Devisces and Pseudodevices.) The items of an Attach Table entry are as follows:

1) right_relp - is a relative pointer to the next entry at the level of the current entry if one exists, else to the parent entry at the next higher level if one exists, else null;

2) down_relp - is a pointer to the first entry at the next lower level if one exists, else null;

3) up_relp - is a pointer to the parent entry at the next higher level if one exists, else null;

4) ioname1 - is the ioname specified by the first argument of the user's attach-call. It serves as the name of the entry and is the key on which the Attach Table is searched;

5) proc_id - is the identification of the process to which the process-independent items of the entry are relevant;

6) type - is the type specified by the second argument of the user's attach call. It is used by the Not Founder (see MSPM Section BF.2.12) to search the Type Table which relates each type to a unique outer module (see MSPM Section BF.2.14, The Type Table Maintainer);

7) ioname2 - is the (third) argument of the user's attach call, It is used by outer modules either to identify a medium (tape reel, file system file, etc.) or to specify another ioname to which the current one is related in a way specified by the type;

8) ttentry_relp - is a pointer to the Type Table entry which was in effect at the time of attachment of the ioname.

9) ttentry_table - records the table (local = 1, global = 0) in which the type table entry was found when ioname1 (above) was attached. This is used to develop a pointer to the correct type table entry when ioname1 is referenced for the first time by processes other than the attaching process;

10) valid_level - is the current value of the validation level for the ioname1.

11) iosegname - is the name of the per-ioname segment for ioname1, and is formed as follows:

```
dcl iosegname char (50),
    ioname1 char (32),
    unique_chars ext entry returns (char(15));
    iosegname = ioname1||"ion"||unique_chars("0"b);
```

12) segp - is a pointer to the per-ioname segment for the ioname of the current entry;

13) auxptr - is an auxilliary pointer for the use of the outer module which recevies control on outer calls for current ioname. It is null by default;

14) tbindex - is a relative pointer to the head (most recently allocated block) of the chain of transaction blocks associated with the transactions on the ioname of the current entry;

15) epvptr - is a pointer to the entry-point vector of the outer module to which control is forwarded by the I/O Switch when it receives outer calls specifying the ioname of this entry;

16) entry_mask - Each bit position corresponds to an outer call. If "1"b then the outer module of the previous item has an entry_point for the call; if "0"b it does not;

17) dtabN - is a pointer to a segment containing a driving table for the servicing outer module;

18) new_dtab - is turned on (set = to "1"b) when a type table entry is edited to alter a driving table. When ioname1 is referenced , the on-condition signals the switching complex that the new driving table must be made known (by calling the SMM) to the process in which it is running;

20) next_vector - is an array of relative pointers to other entries in the AT whose ionames have followed the current ioname as switchpoints in some iopath. This item and the next are explained in detail in the paragraph on the ATM search Algorithm;

21) next_vector.size - is the number of elements of the next_vector which have been used in contrast to the number (10) declared;

22) next_vector.relp - is an array of relative pointers to "probable" next ionames.

ATM Inner Calls

The following is a description of all ATM inner calls, that is, ATM calls whose use is not restricted to the Switching Complex.

1) call atm$get_iosegname(ioname,iosegname,cstatus);

```
        dcl ioname char(*),
            iosegname char(*),
            cstatus bit(18);
```

This call returns the (file system) entry name of the per-ioname segment.

2) call atm$get_pibp(ioname,pibp,cstatus);

```
        dcl ioname char (*),
            pibp ptr,
            cstatus bit (18);
```

This call returns a pointer to the per-ioname segment associated with ioname.

3) call atm$get_valid_level(ioname,valid_level,cstatus);

```
        dcl ioname char(*),
            valid_level fixed,
            cstatus bit(18);
```

This call returns the validation level number for ioname.

4) call atm$set_ioname1(oldname,newname,cstatus);

```
        dcl oldname char(*),
            newname char (*),
            cstatus bit(18);
```

This call changes the name of the Attach Table entry oldname to newname.

5) call atm$set_valid_level(ioname,valid_level,cstatus);

```
        dcl ioname char (*),
            valid_level fixed,
            cstatus bit(18);
```

This call sets the validation level number for ioname to valid_level.

6) call atm$change_dtab(ioname,dtabn,dtabname,dir,copysw,
        offset,cstatus);

```
          dcl ioname char(*),
```

```
          dtabn fixed bin,
          dtabname char(*),
          offset fixed bin,
          copysw bit(1),
          cstatus bit(18);
```

This call changes the driving table pointer number _dtabn_ used by ioname _ioname_ to the file named _dtabname_ with offset _offset_.

7) call atm$switch_ionames(ionamea,ionameb,cstatus);

```
     dcl ionamea char(*),
         ionameb char(*),
         cstatus bit(18);
```

This call exchanges the ionames of the two nodes.

8) call atm$attach_return(ioname,type,ioname2,status);

```
     dcl ioname char(*),
         type char(*),
         ioname2 char(*),
         status bit(144);
```

This call establishes an AT entry and a per-ioname segment for _ioname_, but does not propagate an _attach_ call.

9) call atm$rename_attach_return(oldname,newname,
type,ioname2,status);

```
     dcl oldname char(*),
     newname char(*),
     type char(*),
     ioname2 char(*),
     status bit(144);
```

This call renames _oldname_ to _newname_ and then _attach_es a new node with name _oldname_.

10) call atm$group_init;

Called by io_ctl$init in the Overseer.

11) call atm$queue_restart(ioname,cstatus);
dcl ioname char(*), cstatus bit(18);

There is a delayed_restart bit in the per-I/O segment header. When this call is made, the ATM calls the Locker to try to lock the I/O segment. If the lock attempt succeeds, the ATM passes a _restart_ outer call. Otherwise, it sets the delayed_restart bit in the header (ignoring the lock). When a return is made to the switch, the IOSW checks the bit and, if it is ON, _restart_.

12) call atm$delete_ioname(ioname,delay_sw,cstatus);

```
dcl ioname char(*),
    delay_sw bit(1),
    cstatus bit(18);
```

This call deletes the AT entry for ioname as well as the per-ioname segment.

The Attach Table Search Algorithm

As described in Section BF.2.10, the IOSW knows the outer module to be called by meansof the ioname/outer-module correspondence embedded in the AT. That is, each time the IOSW receives an outer call referencing a given ioname, the IOSW must have access to certain items of the AT entry corresponding to the referenced ioname. One way to obtain these items is by a search of the AT (by the ATM) using the ioname as a key. But this does not avail itself of vital information which all but makes a search, as such, unnecessary. The key observations to make are:

1) generally, the number of switch points (ionames) which follow a given ioname in the iopath is small;
2) the predecessor/small-number-of-successor switchpoint relations may be discovered dynamically and embedded in the AT;
3) the proper relation may be made available to the ATM at the time the ATM has to search the AT, and therefore the standard search need only be invoke as a kind of ioname-not-in-next-list faultcatcher.

Consider Figure 3, with the following definitions:

NXTV(X) (the next vector associated with the ioname X) is an array of (relative) pointers in the AT(X) (the attach table entry for X);

NXTVP (next vector pointer) is an internal static pointer variable of the IOSW;

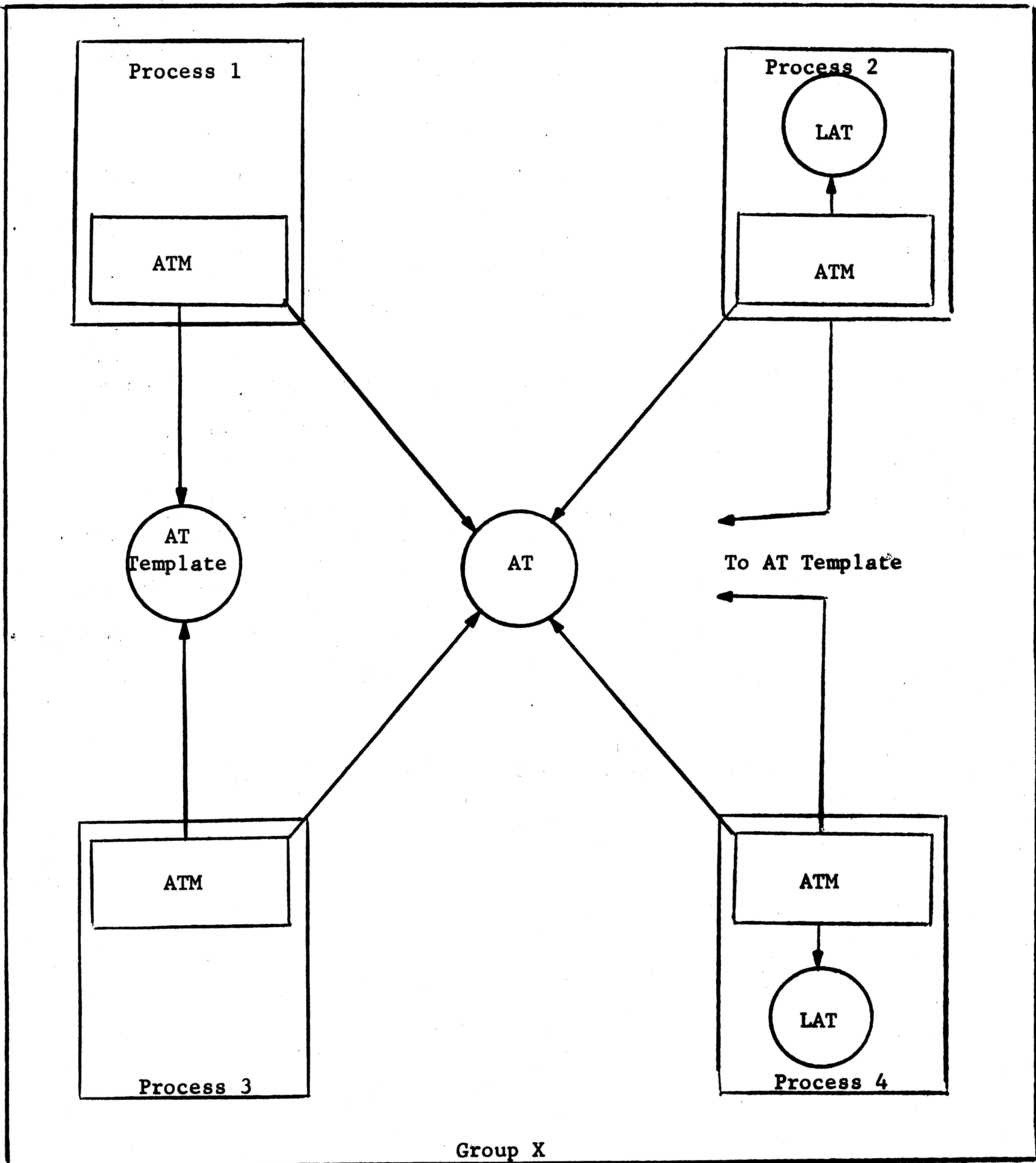
SNXTVP (save next vector pointer) is an automatic pointer variable of the IOSW;

Assume the IOSW is in control at a point after the ATM has found the AT(A) and returned the values of the required items. Hence, the IOSW may set NXTVP equal to the address of NXTV(A). At some point after this, the IOSW calls the target outer module OM(A). Let OM(A) now issue an outer call referencing the ioname B. Receiving control, the IOSW saves NXTVP in SNXTVP and issues a call to the ATM passing NXTVP and the ioname (B) as arguments. The ATM attempts to find a match for B by comparing it with the ionames of the AT entries whose addresses are contained in the NXTV pointed to by NXTVP. Assume this attempt fails, then the ATM invokes an exhaustive search of the AT to find the AT entry for B. Assuming there is no error, the exhaustive search must succede The ATM then updates the NXTV pointed to by NXTVP with a

pointer to the found AT entry for B. (On future calls referencing B by OM(A), the NXTV(A) will contain a pointer to B, hence the exhaustive search will not be necessary.) At this point, whether or not the exhaustive search was invoked, the ATM sets NXTVP equal to the address of NXTV(B) and sets the other return values required by the IOSW and returns. When the IOSW receives the return from the ATM it, it saves NXTVP in SNXTVP (as above) and calls OM(B). When the IOSW receives the return from OM(B) it sets NXTVP equal to SNXTVP and returns. Because of the saving and restoring of NXTVP, the value of NXTVP is correct no matter how complex the iopath. Finally, note that if NXTVP is initialized to point to a NXTV not associated with any ioname, then that NXTV will build up to the list of ionames defined by the user, and not contain any ionames created by the IOS in setting up iopaths. This will improve the search time even in the case in which a next vector does not at first seem appropriate, that is, at the user/IOSW interface.

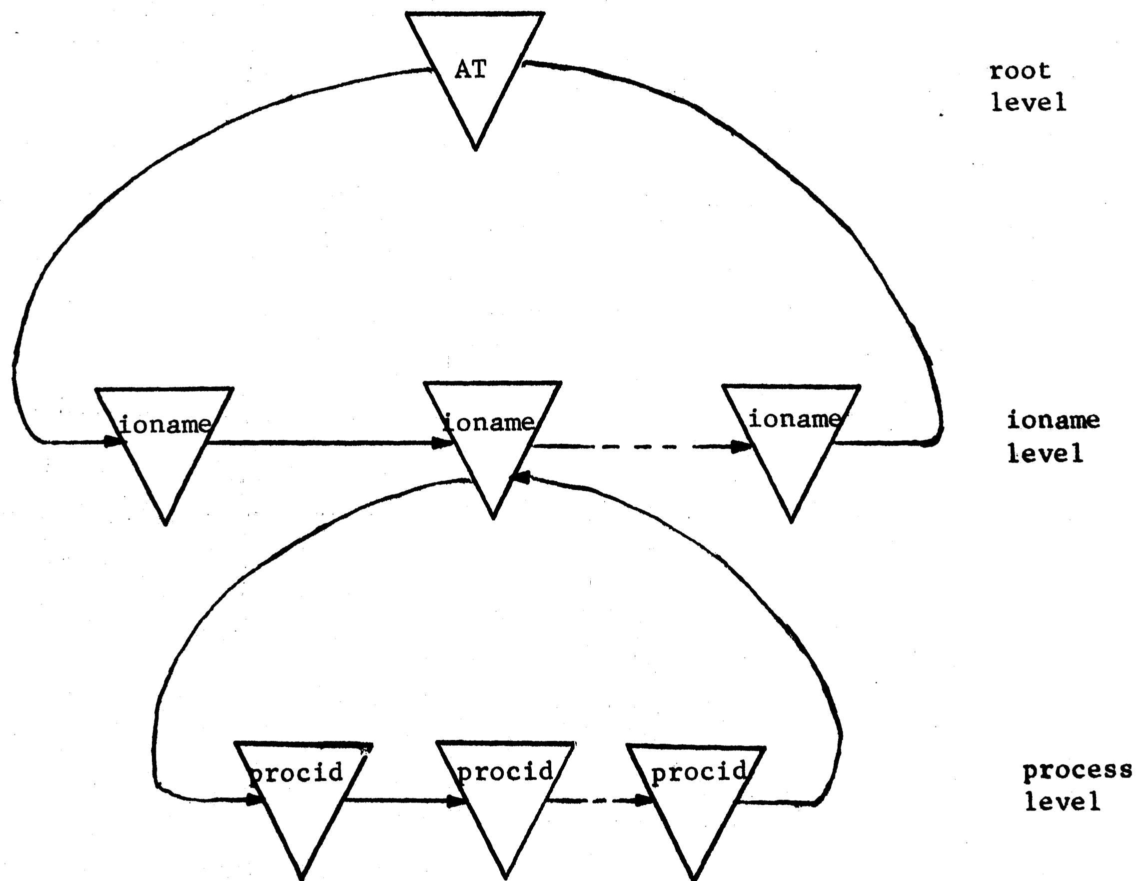
```
dcl 1 at_entry based (p),        /*AT entry*/
 2 right_relp bit(18),           /*PIG, relp to next entry at this level*/
 2 down_relp bit (18),           /*IG, relp to first lower level entry*/
 2 up_relp bit (18),             /*PIG, relp to higher level entry*/
 2 ioname1 char (32),            /*I, primary ioname*/
 2 proc_id bit (36),             /*P, process id*/
 2 type char (32),               /*I, attachment type name*/
 2 ioname2 char (32),            /*I, secondary ioname*/
 2 ttentry_relp bit(18),         /*I, relp to type table entry active
           "                       at time of attach*/
 2 ttent_table bit(1),           /*"1"b if ttentry is local, else "0"b*/
 2 valid_level fixed,            /*validation level*/
 2 iosegnam char (50),           /*I name of per-ioname segment*/
 2 segp ptr,                     /*P, ptr to per-ioname segment*/
 2 auxptr ptr,                   /*P, outer module auxiliary ptr*/
 2 tbindex bit (18),             /*P, index of last allocated TB*/
 2 epvp ptr,                     /*P, ptr to entry point vector*/
 2 entry_mask bit (72),          /*PI, entry point mask*/
 2 dtabp1 ptr,                   /*P, driving table pointers*/
 2 dtabp2 ptr,
 2 dtabp3 ptr,
 2  new_dtab bit (1),            /*P, new driving table flag*/
 2 next_vector,                  /*I, data base for predictive search
           "                       of next AT entry*/
  3 size fixed,                  /*I, number of AT entries for all
           "                       known next ionames*/
  3 relp (10) bit (18),          /*I, relative pointers to AT entries*/
 2 flags,                        /*I, noattach/local flags*/
  3 noattach bit (1),
  3 screen bit (1);
```

<u>Table 1 - Attach Table Declaration</u>

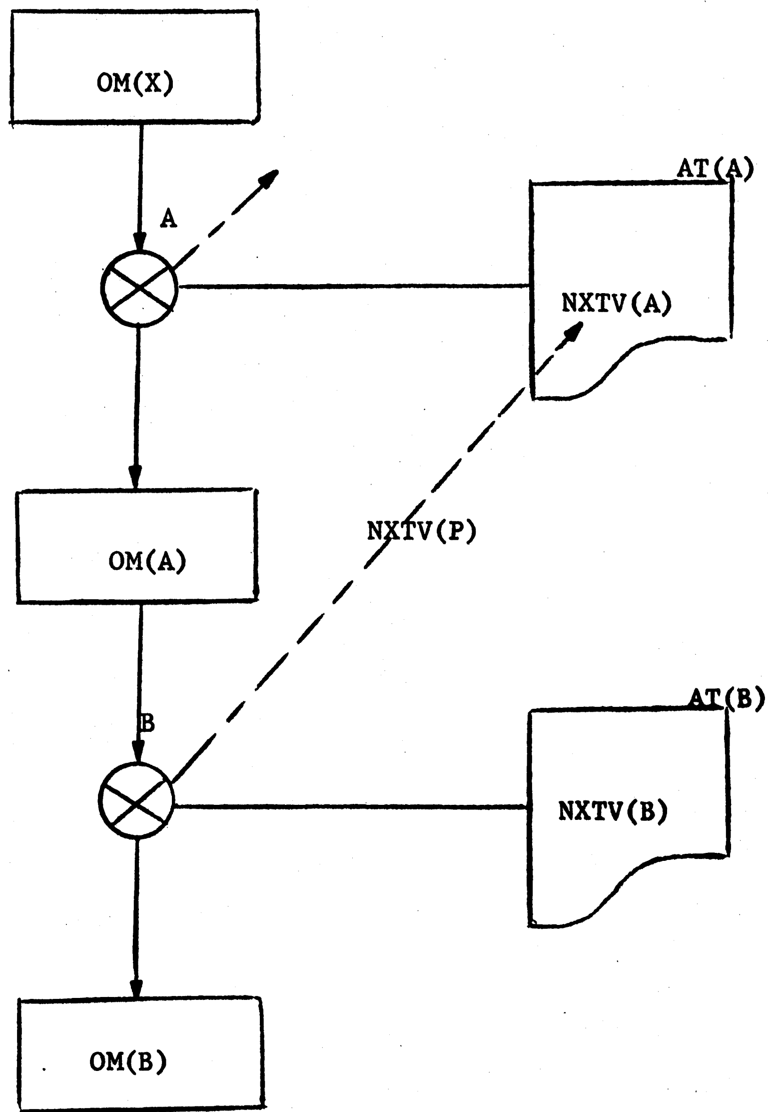


Attach Table Location

Figure 1



Attach Table Structure
Figure 2



Attach Table Search Algorithm

Figure 3