To: MSPM Distribution
From: J. F. Ossanna
Subj: BF.2.20
Date: 1/10/68

In addition to minor corrections, the attached revision of BF.2.20 contains the following changes.

1. New declarations are given for the per-ioname-segment header, the per-ioname base (PIB), and the transaction block segment (TBS).

2. The description of the use of the Locker by the I/O switch and the Transaction Block Maintainer (TBM) is expanded.

3. The initialization of the TBM and the TBS is described. A new call, tbm$init, is specified.

4. A new call to unthread transaction blocks from down chains, tbm$unthread, is described.

5. The status bits returned by the TBM are specified.

Published: 1/10/68
(Supersedes: BF.2.20, 8/14/67)

Identification

Data Base and Transaction Block Discipline for Outer Modules.
J. F. Ossanna.

Purpose

This section describes the data base discipline for I/O System (IOS) outer modules. With few exceptions, all nonautomatic data is kept in per-ioname Segments (IS). Transaction Blocks (TB's) are kept in a per-user-group Transaction Block Segment (TBS). The use of driving tables by table-driven modules is described. The use of the Transaction Block Maintainer (TBM) is described. A data buffer discipline is described which makes use of standard transaction block manipulation.

General

At the time an ioname is defined as a result of an attachment, the switching complex creates a per-ioname Segment (IS) and establishes certain data bases within it. One of these data bases, called the Per-ioname Base (PIB), becomes the principal data base of the outer module reached via the ioname.

For every outer call reaching the outer module (except divert), the I/O switch allocates a Transaction Block (TB) within the Transaction Block Segment (TBS) by calling the Transaction Block Maintainer (TBM). These blocks are chained together in a way described later in this document, and indices identifying the chain are stored within the PIB. The contents of these blocks are accessible by calls provided by the TBM. Upon request, the TBM provides chaining of related blocks; for example, the blocks for outgoing calls made by an outer module as a result of an incoming call may be included in the "down chain" of the block for the incoming call.

Additional per-transaction data is kept in Transaction Block Extensions (TBE's) allocated within an area in the PIB by the outer module itself; a relative pointer to the TBE is kept in the corresponding TB. Additional per-ioname data is kept in similarly-allocated PIB Extension (PIBE's).

Using a system involving several hold bits representing various interests, the TBM provides automatic deallocation of unwanted TB's together with their TBE's.

Each outer module may create additional transaction block chains for internal purposes. For example, the buffering of data being processed by the IOS is implemented by such a chain. The outer module allocates (by a call to the TBM) a transaction block to which the buffer is appended in the form of a transaction block extension.

Relative Pointer Use

Once an ioname is known to a user process group, a procedure in any of the processes in the group can issue an outer call directed at that ioname. Thus an iopath is exercisable within any process in the group. As a consequence, pointer information cannot conveniently be stored in the various common data bases in the form of pointer variables. The doctrine on pointer information is that such information is stored as "relative pointers"; these are bit strings of length 18, and are easily converted to and from ordinary pointers by use of the procedures described in Section BY.14. In variable names and declarations throughout this section, "relp" is commonly used to indicate a relative pointer.

Pointer variables which are not easily computed by an outer module are provided by the switching complex which determines and keeps them for every process using an ioname. Examples are the pointer to the PIB itself and pointers to driving tables.

The Per-Ioname Segment

The per-ioname Segment (IS) is created by the switching complex at the time an ioname is first defined as a result of an attach call. The IS then continues to exist until the outer module associated with the ioname is successfully detached. That outer module initiates the deletion by issuing an atm$delete_ioname call requesting delayed deletion; both the ioname and the IS are deleted upon return by the I/O switch.

The IS contains a number of separately-allocated data bases. All of these data bases are accessed as PL/I "based" structures. The data bases in the IS are:

1. Header; created and initialized by the switching complex.
2. Standard Per-Ioname Base (PIB); allocated by the switching complex.
3. Standard PIB auxiliary based storage; allocated by the outer module.
4. PIB extensions (PIBE's); allocated by the outer module.
5. Transaction block extensions (TBE's); allocated by the outer module.
6. Interprocess communication block (ICB); allocated by the attachment module in a Device Strategy Module (DSM) (see Section BF.2.23).

All data bases allocated by the outer module are allocated within a PL/I area in the PIB. The header, PIB, the PIB's auxiliary based storage, and the ICB have standard declarations which are

given later in this section. Although the PIB and TBE extensions are intended for outer-module-dependent data, a portion of their declarations is standardized to enforce a standard chaining of these extensions.

The standardized chaining of extensions permits an outside procedure -- e.g. a dump routine -- to determine completely where all the data in the IS is.

The IS Header

The IS header contains process-independent per-ioname data of interest primarily to the I/O switch. In particular, an outer module need not be concerned with the header. The only exception to the latter statement occurs in a DSM in which the attachment module (see Section BF.2.23) and the request queuer (see Section BF.2.24) use information in the header.

The header is created and initialized by the switching complex at the time the IS is created. The declaration for the header follows.

```
dcl 1 ioseg based (p),          /*per-ioname segment header*/
 2 flag,
  3 delayed_detach bit (1),     /*if ON, IOSW deletes ioname */
  3 restart bit (1),            /*if ON, IOSW issues restart call*/
 2 relp,                        /*relative pointers*/
  3 pib bit (18),               /*relp to PIB*/
  3 icb bit (18),               /*relp to ICB*/
 2 lock bit (144),              /*standard lock structure space*/
 2 dummy ptr;                   /*dummy element for PIB origin*/
```

The "delayed-detach" switch is used by the switching complex to implement delayed deletion of the ioname and the IS. The "restart" switch is used to implement a part of the restart strategy (see Section BF.2.23-25). The relative pointer to the PIB is used by the I/O switch to supply the outer module with a real pointer to the PIB (see discussion below). The relative pointer to the ICB is used by the attachment module and the request queuer in a DSM.

The lock structure is used by the I/O switch when calling locker$wait to lock the ioname (see Section BQ.7.00). When the switch receives an outer call, it attempts to lock the attachment graph node corresponding to the ioname by calling the Locker; an argument in the call is a pointer to the lock structure. The ioname remains locked until the switch is about to return to the original caller, at which time the lock is released by another call to the Locker. This ioname locking strategy prevents simultaneous use of an ioname and the corresponding per-ioname segment of an iopath by procedures in more than one process (see BF.1.03 and BF.2.11 for additional discussion).

The Per-ioname Base (PIB)

The Per-ioname Base (PIB) is the principal data base of an outer module. The PIB is allocated and initialized by the switching complex at the time the per-ioname segment (IS) is created. The PIB is accessed as based storage by means of a pointer supplied by the I/O switch with every call. Additional per-ioname data is allocated by the outer module in the form of PIB extensions (PIBE's); see discussion later in this section.

The pointer to the PIB (pibp) is supplied by the I/O switch to the outer module as an additional last argument on every outer call routed to the outer module. In addition, the I/O switch initializes two groups of PIB items prior to routing each call. One group consists of all pointer items; the second consists of certain items which are copied from the caller's PIB. This per-call initialization is discussed in detail below.

The contents of the standard PIB is chosen to accommodate the common needs of a majority of outer modules. The PL/I declaration follows.

```
dcl 1 pib based (p),          /*standard PIB*/
 2 sync_event bit (70),       /*for sync management*/
 2 error_event bit (70),      /*  "  */
 2 dtabp1 ptr,                /*driving table ptr 1, mode control*/
 2 dtabp2 ptr,                /*driving table ptr 2*/
 2 dtabp3 ptr,                /*driving table ptr 3*/
 2 auxptr ptr,                /*auxiliary outer module ptr*/
 2 ioname1 char (32),         /*ioname1*/
 2 typename char (32),        /*type name*/
 2 ioname2 char (32),         /*ioname2*/
 2 bmode bit (72),            /*mode bit string*/
 2 next_ioname char (32),     /*next ioname*/
 2 nmore fixed bin,           /*number of additional next ionames*/
 2 elsize fixed bin,          /*element size*/
 2 readbit fixed bin,         /*read pointer in bits*/
 2 writebit fixed bin,        /*write pointer in bits*/
 2 lastbit fixed bin,         /*last pointer in bits*/
 2 boundbit fixed bin,        /*bound pointer in bits*/
 2 nbrk fixed bin,            /*number of break delimiters*/
 2 ndelim fixed bin,          /*number of read delimiters*/
 2 relp,                      /*relative pointers*/
  3 pibe bit (18),            /*relp to PIB extension*/
  3 more bit (18),            /*relp to additional ionames*/
  3 brk bit (18),             /*relp to break list*/
  3 delim bit (18),           /*relp to delimiter list*/
 2 chain_base,
  3 t1index bit (18),         /*base of call transaction block chain*/
  3 t2index bit (18),
  3 b1index bit (18),         /*base of buffer chain*/
  3 b2index bit (18),
  3 a1index bit (18),         /*base of aux transaction block chain*/
```

```
  3 a2index bit (18),
  2 ioarea area ((MAX));          /*outer module allocation area*/
```

The sync_event and the error_event are used for internal synchronization management (see Section BF.2.02); modules not concerned with such matters can ignore these items except for one requirement. These items, when present (not zero), must be passed along to the next outer module. The mechanism for doing this consists of the module adding its "pibp" as an extra argument on outer calls it issues. Upon receiving an outer call with a "pibp", the I/O switch copies these two items into the callee's PIB before passing on the call. Thus outer modules must always include their "pibp" as an extra argument on every call issued to the next outer module in the iopath. If the I/O switch does not receive a "pibp", it zeros these items in the callee's PIB. Thus the I/O switch provides automatic forwarding of these synchronization management items.

There are two exceptions to the rule that an outer module must include its "pibp" in its calls. First, the module in the iopath that is the ultimate recipient of these items (the DSM) does not forward them. Second, the "pibp" must not be included on calls representing "incidental" input/output, i.e. calls issued to an ioname which is not a next ioname along the module's iopath. An example of incidental input/output is writing an error comment on a standard ioname used universally for that purpose. The use of such incidental input/output by outer modules contains an element of danger (e.g. the standard ioname may have been attached to the iopath containing the module) and the circumstances under which is may be permitted are not yet determined.

The driving-table pointers and the auxiliary pointer are initialized by the I/O switch on every call, so that they are appropriate to the process in which the call is being made. The driving table pointers point to the first word of the segments containing the driving tables. The pointers are determined by the switching complex from driving table names kept in the Type Table (see Section BF.2.14). This mechanism is provided to permit outer modules to be table-driven where possible. One kind of driving table used by all outer modules is the mode control structure (see Section BF.2.27); dtabp1 in the PIB is reserved for this table. The code conversion tables and the code conversion module are a good example of the table-driven module approach.

The auxiliary pointer, "auxptr", is used to reference an external segment known only to the outer module. Such use of an external segment is permitted only in approved cases. One case involves the use of the I/O Registry Files by the DSM and the DCM. Another example of an external segment is the common data base shared by DCMs which operate devices connected to shared channels. The File System Interface Module uses auxptr to access the File System file. Since the switching complex does not know the identity of the external segment, it is up to the outer

module to compute auxptr. Upon a return from the outer module the I/O switch saves the value of this pointer in a per-process entry of the Attach Table. On subsequent calls to the module, the switch restores the pointer to the value previously saved for that process. If upon return the I/O switch notes that a restored auxptr has been modified by the outer module, the values saved for the other processes are made null. If no previous value had ever been saved, a null pointer is used for the initial value. Outer modules using auxptr should always test for a null pointer before the first use during a call.

"ioname1", "typename", and "ioname2" correspond to the first three arguments of the original attach call received by the outer module. Their values are assigned by the outer module at attachment time. "bmode" is the mode string array returned by the Mode Handler (see Section BF.2.27).

After the outer module determines by some algorithm what the ioname of the next module in the iopath is, this ioname is stored in "next_ioname". When more than one next ioname is involved (such as in the case of a broadcaster), the additional ionames are kept in auxiliary based storage to be described below. The number of additional ionames is kept in "nmore".

The current element size measured in bits is kept in "elsize". The various "pointers", the read, write, last and bound pointers, are kept as bit counts. The determination of these pointers as element counts is always accomplished by dividing the bit counts by the current element size.

"nbrk" and "ndelim" are the current numbers of break and read delimiter elements respectively. The actual strings of these elements are kept in auxiliary based storage to be described below.

The relative pointer "pibe" points to the first PIB extension; "more", "brk", and "delim" point to the auxiliary based storage for the additional ionames, the break element string, and the read delimiter string, respectively. The declarations for the auxiliary storage follow.

```
dcl brklist (nbrk) bit (elsize) based (p),  /*p related
          "                                  to relp.brk*/
 delimlist (ndelim) bit (elsize) based (q), /*q related
          "                                  to relp.delim*/
 more_ionames (nmore) char (32) based (r);  /*r related
          "                                  to relp.more*/
```

This auxiliary storage is allocated within "ioarea" by the outer module whenever necessary. If any of this auxiliary storage is not needed, the corresponding relative pointer should be zero.

The various transaction block chain base indices are discussed later in this section.

All based storage allocated by the outer module is allocated within the PL/I area, "ioarea".

PIB Extensions

Per-ioname data not resident in the PIB are kept in what are known as PIB Extensions (PIBEs). There can be any number of PIBEs. Successive PIBEs are reached by relative pointers kept in immediately-preceding PIBEs. Each PIBE is required to contain an included measure of its size. The standard declaration for any PIBE follows.

```
dcl 1 pibe based (pn),          /*standard PIBE form*/
 2 relp,                        /*standard PIBE chaining*/
  3 next bit (18),              /*relp to next PIBE*/
  3 last bit (18),              /*relp to last item in this structure*/
        ...       ;             /*outer module's private dcls*/
```

The "next" is a relative pointer to the next PIBE in the chain. "last" is a relative pointer to the last item in the PIBE structure, and is included to permit size determination by outside procedures. The relative pointer to the first PIBE is (relp.pibe) in the PIB. The last PIBE must have (relp.next) zero. An outer module can avoid chasing a PIBE chain by keeping a copy of all PIBE relative pointers in the first PIBE.

Examples of PIB and PIBE Use

The real pointers corresponding to the relative pointers kept for accessing IS based storage must be computed each time a new call is made to the outer module. The real pointers therefore exist only in automatic storage. For example, to obtain the pointer p1 for use in accessing the first PIBE, one of the pointer manipulation procedures of BY.14 is used:

```
p1 = ptr$ptr(pibp,pibp->pib.relp.pibe);
```

The code to freshly allocate a second PIBE is:

```
allocate pibe in (pibp->pib.ioarea) set (p2);
p1->pibe.relp.next = ptr$rel(p2);
p2->pibe.relp.last = ptr$rel(addr(p2->pibe.last_item));
p2->pibe.relp.next = "0"b;
```

The code to allocate a new list of break elements following a setdelim call is:

```
brkp = ptr$ptr(pibp,pibp->pib.relp.brk);
free brkp->brklist;
allocate brklist in (pibp->pib.ioarea) set (brkp);
pibp->pib.relp.brk = ptr$rel(brkp);
```

The Interprocess Communication Block

The Interprocess Communication Block (ICB) is a common data base for the attachment module, the DSM's request queuer, and the device manager dispatcher. The ICB is allocated by the attachment module. For a discussion of the purpose and use of the ICB, see Sections BF.2.23, BF.2.24, and BF.2.25. The declaration for the ICB may be found in both BF.2.23 and BF.2.25.

Introduction to Transaction Block Discipline

The creation of Transaction Blocks (TBs) for outer calls (and for certain buffering functions and for nonswitched iopath-directed calls) together with the ability to associate related blocks, is the basic mechanism for maintaining the necessary history of individual transactions and for preserving the relationships between these transactions.

A transaction block is a short fixed-length structure containing mostly information relating the block to other related blocks. The data in the block of direct interest to an outer module is a status bit string, some flags, and a relative pointer to a Transaction Block Extension (TBE). One such block is allocated by the I/O switch (by a call to the Transaction Block Maintainer) every time an outer call is made (except for the divert call); this block is automatically chained into the chain of blocks corresponding to calls issued to the same ioname. The base of this chain is anchored in the callee's PIB; specifically, "t1index" and "t2index" are actual indices into a transaction block array and point to the oldest and newest end of the chain respectively. Further, this chain essentially belongs to the callee; the only item in a block of interest to the caller is the status bit string.

Additional per-transaction data is kept in TBEs allocated by the outer module. The standard portion of the declaration of a TBE is exactly the same as that described earlier for a PIBE. Any number of TBEs may be chained; the relative pointer to the first is kept in the parent TB.

Provision is made in the PIB for basing two additional TB chains. Modules retaining user data between calls must use a standard buffer chain. The module allocates (by a call to the Transaction Block Maintainer) a TB for every buffer needed; the actual data is kept in a corresponding TBE. Buffering is discussed in more detail later in this section. A third chain, called the auxiliary chain, is used by modules making outgoing calls but having no outgoing switch node. The auxiliary chain is used by DCMs to maintain the necessary per-transaction data for calls to the GIOC Interface Module (GIM), and is used by the DSM's Request Queuer to communicate with the Device Manager's Driver.

In addition to the chaining of the three "main" TB chains mentioned above, the Transaction Block Maintainer (TBM) will, upon request, create secondary chains of related blocks. The secondary chains are known as "down" chains and consist of

existing blocks in main chains. The base of a down chain is anchored in some TB. A block can be included in the down chain based in some other block by a call to the TBM. Typical usage is to include blocks for outgoing calls in the down chains of blocks for corresponding incoming calls in such a way that status may be easily updated. An example of such use is given later. When buffer chains are in use, the incoming TBs' down chains include the corresponding buffer blocks, and each buffer block's down chain includes the corresponding outgoing TBs.

All the transaction blocks for a user are kept in a Transaction Block Segment (TBS). All use of these blocks by outer modules is by calls to the Transaction Block Maintainer (TBM).

The lifetime of a transaction block is controllable. A mechanism involving several hold bits representing various interests is used to delay the otherwise automatic deallocation of blocks. The holding and releasing of blocks is discussed later below.

The Transaction Block Maintainer (TBM) services the allocation and deallocation of Transaction Blocks (TBs), the chaining of related blocks, the chasing of these chains, the storing and retrieving of the transaction status, outer module flags, and the TBE relative pointer. It also services the setting and resetting of the hold bits which control deallocation.

The Transaction Block Segment

The Transaction Block Segment (TBS) contains all the transaction blocks for a user process group and is the principle data base of the TBM. The declaration for the TBS follows.

```
dcl 1 tbs based (p),          /*transaction block segment*/
 2 lock bit (144),            /*standard lock structure space*/
 2 index,                     /*indices*/
  3 vacant1 bit (18),         /*head of vacant list*/
  3 vacant2 bit (18),         /*tail of vacant list*/
  3 orphan1 bit (18),         /*head of orphan list*/
  3 orphan2 bit (18),         /*tail of orphan list*/
  3 last bit (18),            /*highest block used*/
  3 pad bit (18),             /*padding*/
 2 tb (MAX),                  /*transaction blocks*/
  3 status bit (144),         /*outer call status*/
  3 hold bit (6),             /*hold bits*/
  3 tbm_flags bit (6),        /*flags for TBM*/
  3 om_valid bit (6),         /*outer module validation level*/
  3 flags bit (18),           /*outer module flags*/
  3 chain_cnt bit (18),       /*chain inclusion count*/
  3 tberelp bit (18),         /*relp to TBE in per-ioname segment*/
  3 xn1 bit (18),             /*main chain next index; from x1*/
  3 xn2 bit (18),             /*main chain next index; from x2*/
  3 dn1 bit (18),             /*down chain next index; from d1*/
  3 dn2 bit (18),             /*down chain next index; from d2*/
```

```
   3 d1 bit (18),              /*down1 index*/
   3 d2 bit (18);              /*down2 index*/
```

The lock structure is used by the TBM to lock the TBS whenever necessary by calling locker$wait (see Section BQ.7.00); the TBM waits for the TBS to become free. The transaction blocks are members of a transaction block array; when the TBM is requested to "allocate" a block, it merely obtains a currently unused block from the vacant list. When a block is deallocated it is returned to the vacant list. "vacant1" and "vacant2" are the indices to the head and tail respectively of the vacant list. "orphan1" and "orphan2" are the indices to the head and tail respectively of the "orphan" list. The orphan list contains blocks which have been removed from their original main chains but which do not yet meet the full conditions for deallocation; this mechanism is discussed later below.

Each transaction block contains a standard status bit string, a string of hold bits, a string of flag bits private to the TBM, a string of flag bits usable by the outer module, the outer module's validation level, a chain inclusion count, a relative pointer to the transaction block extension, and indices of related transactions. Most of these items are discussed in detail later below. The outer module's bit string (flags) is provided solely for the module's arbitrary use; for example, it may be used to conveniently differentiate between useful transaction categories. The TBM's bit string (tbm_flags) is used to indicate whether a block is in a main chain, in the orphan list, or in the vacant list. The outer module validation level (om_valid) is used by the TBM to control what procedures can call to have certain items in the block set.

Transaction Block Allocation

The following call to the TBM allocates a new block in a main chain.

```
call tbm$allocate(chain_base_ptr,holdn,tbindex,cstatus);

   dcl chain_base_ptr ptr,   /*base of main chain*/
    holdn fixed bin,   /*hold bit indicator*/
    tbindex bit (18),  /*TB index of new block*/
    cstatus bit (18);  /*allocate call status, see Table 1*/
```

The chain base ptr is a pointer to a pair of chain base indices in the outer module's PIB. If the indices x1 and x2 point to the oldest and newest block respectively in the x chain, the TBM allocates the new block in the x2 end of the chain. Here "x" corresponds to the "t", "b", or "a" in the chain-base-indices' names in the PIB. The index for the newly-allocated block is returned in tbindex and can also be found in the location for x2. holdn permits the caller to set the nth hold bit; if holdn is zero, the hold bits are initalized to zero. See discussion on holding later below.

Transaction blocks for outer calls are allocated by the I/O switch. When the outer module receives control, it can find the index to the block corresponding to the current call in (chain_base.t2index) in the PIB. However, the status bit string is initialized by the I/O switch and bits 127-144 of the status string also contain the new TB index (bits 1-126 are initialized zero). A convention of using the value of the TB index extracted from the status string removes the necessity of coping with the effect of recursive entry upon the value of "t2index". (Although the ioname is locked upon first entry, a recursive entry by the same process is permitted; see Section BF2.11).

Transaction blocks for buffer or auxiliary chains are allocated by tbm$allocate calls issued by the outer module itself.

When the TBM freshly allocates a block, it initializes om_valid to zero. Upon receipt of the first subsequent call which requires storing information in the block, the TBM stores the current validation level in om_valid. Further such calls are fulfilled only if the caller's validation level is equal to or less than that in om_valid.

Transaction Block Holding

The "hold" bit string in a transaction block contains six hold bits called hold1,...,hold6. Setting any of these bits nonzero prevents deallocation of the block. At present only hold1, hold2, and hold3 are assigned. With respect to an ioname, hold1 is for the caller and hold2 is for the callee. In those cases where a block is known to only one module (e.g. buffer blocks), hold2 is used by that module. Hold bit hold3 is used by the I/O switch to guarantee a caller a chance to set hold1 (see Section BF.2.11 for a detailed explanation).

The caller may wish to hold certain blocks in order to later examine an updated version of the status bit string. The status originally returned to a caller contains the then current status in bits 1-126 and contains a transaction block index in bits 127-144. During later calls, the callee updates the status of earlier calls when appropriate and if their blocks still exist. Further, it is the status string in the block which is updated, not the original caller's status string. The following calls are used by the caller to set and reset hold1.

```
call hold(status,cstatus);

call release(status,cstatus);

     dcl status bit (144), /*returned outer call status*/
      cstatus bit (18);  /*hold/release call status, see Table 1*/
```

The TB index in the status argument is used by TBM to identify the correct block. A related TBM call is the following.

```
call getstatus(status,cstatus);
```

This call is used to replace an old status string by a new, possibly updated one. The caller provides status equal to the status bit string of a previous transaction which is being held; the TBM uses the TB index provided in status to identify the block and returns status equal to that currently in that block. The returned status contains a valid status change indicator (primary status bit 10). The argument declarations are the same as the previous call. See Section BF.1.11 for a detailed discussion of delayed status updating and retrieval, including status change indication. The following calls permit the callee (the outer module receiving the call) to set and reset the hold bits.

```
call tbm$set_hold(tbindex,holdn,cstatus);

call tbm$reset_hold(tbindex,holdn,cstatus);

     dcl tbindex bit (18), /*TB index*/
      holdn fixed bin,   /*hold bit indicator*/
      cstatus bit (18);  /*call status, see Table 1*/
```

holdn is an integer from 1 to 6 indicating which hold bit is to be set or reset in the block whose index is tbindex. If holdn is zero, neither call has any effect. It may be noted that when holdn is one these calls duplicate the functions of the hold and release calls; the latter are designed to be safer and more convenient for the caller to use. If the holdn argument of a tbm$allocate call is zero, the TBM initializes the entire hold bit string to zero.

A block whose hold bits are all zero and whose chain inclusion count is zero is a candidate for deallocation. When tbm$allocate is called, other blocks in the concerned chain are considered for deallocation. When a block is deallocated, it's TBE's are freed and the blocks, if any, included in its down chain have their chain inclusion count (see later below) reduced by one.

Once an outer module is finished with a call block and has had hold2 set to zero, the block will continue to exist in the call chain until the full conditions for deallocation are realized. Inasmuch as an outer module frequently chases this call chain (see below), it would be convenient if only interesting blocks were present. The following call is provided to remove uninteresting but nondeallocatable blocks.

```
call tbm$remove(chain_base_ptr,tbindex,cstatus);
```

tbindex is the index of a block located in the main chain whose base indices are pointed to by chain base ptr. The TBM undertakes the following steps: (1) hold2 is set to zero; (2) the block is deallocated if deallocation conditions are met; (3) if not, the TBEs are freed, the tberelp is set to zero, and the

block is removed from the main chain and placed in the orphan chain. Blocks in the orphan chain can be expected to be eventually deallocated. Whenever an orphan block has any hold bits reset or its chain inclusion count reduced, the TBM considers deallocation.

When an outer module is detached it is necessary to remove all the TBs in main chains based in the module's PIB. Upon receiving a return from a detach call the I/O switch issues the following call on behalf of any such chain whose base indices are not zero.

```
call tbm$delete_chain(chain_base_ptr,cstatus);
```

The TBM performs the functions described for the tbm$remove call for each block in the chain whose base indices are pointed to by chain base ptr.

The Chaining of Related Blocks

Each transaction block contains the indices of the next oldest and next newest block in the same main chain; these are the xn1 and xn2 indices respectively in the block declaration. The oldest block has xn1 zero, and the newest block has xn2 zero. The use of bidirectional next indices permits the main chain to be chased in either direction beginning at any block. We speak, for example, of the xn1 next index as pointing away from the x1 end of the chain or toward the x2 end of the chain, where x1 and x2 are the chain base indices. The top row of blocks in Figure 1 shows the chaining of blocks in a main chain.

A second kind of chaining which is used to associate related blocks is available upon request. Suppose, for example, an incoming call to a module results in three outgoing calls. The incoming call has a transaction block in the module's call chain, and the outgoing calls have blocks in the next module's call chain. It is convenient for future status updating to save the relationship between these blocks. A mechanism is provided which permits threading the blocks for the outgoing calls into a "down" chain based in the block for the incoming call.

The following call threads an existing block into another block's down chain.

```
call tbm$thread(tbindex1,tbindex2,cstatus);

     dcl tbindex1 bit (18),  /*index of TB to be threaded*/
      tbindex2 bit (18),  /*index of TB basing the down chain*/
      cstatus bit (18);   /*thread call status, see Table 1*/
```

tbindex1 is the index of the block to be threaded; the threading will result in its dn1 or dn2 being set nonzero. tbindex2 is the index of the block in which the down chain is or is to be based; the threading will affect the values of its d2 and possibly its d1. Prior to the existance of a down chain based in a block,

both the d1 and d2 down indices are zero; when the down chain exists, d1 is the index of the oldest block threaded and d2 is the index of the newest one. If there is only one block in the down chain, d1 = d2 in the base block, and dn1 = dn2 = 0 in the threaded block. The blocks in the down chain are threaded to each other using the dn1 and dn2 down-chain-next indices; except for blocks at either end of the down chain, dn1 is the index of the next oldest block threaded and dn2 is the index of the next newest block threaded. Except for the blocks at either end of the down chain, a block can be in only one down chain. The block at the d1 end of the chain has dn2 zero if it is in only that chain; if it is also at the d2 end of another down chain, dn2 applies to that chain. A similar situation is true with respect to dn1 for a block at the d2 end of a down chain. A block which is at the d2 end of one down chain and at the d1 end of a second down chain, can be in additional down chains provided it is the only block in those chains. No confusion exists in the interpretation of dn1 and dn2, since the down chains are chased by comparing successive dn1s with d2 (or dn2s with d1). The chain inclusion count (chain_cnt in the TB) is increased by one every time a block is threaded into another down chain.

The second row of blocks in Figure 1 shows a main chain whose blocks are included in down chains based in blocks in the top row. Blocks B1 and B2 are included in the down chain based in block A1; similarly B2, B3, and B4 are included in the down chain based in block A3. Block B2 is in three down chains, those based in A1, A2, and A3.

If desired, a block may be removed from a down chain by making the following call.

```
call tbm$unthread(tbindex1,tbindex2,cstatus);
```

The block whose index is tbindex2 is removed from the down chain based in the block whose index is tbindex2, and its chain inclusion count is reduced by one.

An outer module does not need to be concerned with the effort of chasing main or down chains. The following call is provided for chasing both main and down chains:

```
call tbm$get_chain(tbindex,type,orig,cnt,listptr,cstatus);

     dcl tbindex bit (18),    /*see below*/
      type fixed bin,         /*1=down1, 2=down2, 3=main1, 4=main2*/
      orig fixed bin,         /*offset, see below*/
      cnt fixed bin,          /*size of return list*/
      listptr ptr,            /*ptr to list*/
      1 list (cnt),           /*return list*/
       2 tbindex1 bit (18),   /*TB index*/
       2 flags bit (18),      /*outer module flags*/
       2 status bit (144),    /*transaction status*/
```

```
      2 tberelp bit (18),   /*TBE relp*/
    cstatus bit (18);   /*get_chain call status, see Table 1*/
```

For type = 1 or 2, tbindex is the TB index to the base block containing the down chain to be chased. The down chain is chased from either the d1 end or the d2 end, according to whether type is 1 or 2. For type = 3 or 4, tbindex is an index of a block in a main chain which is to be chased from that point. The main chain is chased from either the x1 end or the x2 end, according to whether type is 3 or 4. orig is the offset from the base block in numbers of blocks. cnt is the number of blocks whose tbindex (tbindex1), flags, status bit string (status), and TBE relp (tberelp) are wanted. A bit in cstatus indicates whether or not there are additional blocks in the chain. If orig = 1, the first block whose data is returned is the first down or next main block; if orig = N (greater than 1), the first data reported is from the Nth down or (next+N-1)th main block; if orig = 0, the first data reported is from the base block itself.

The get_chain call is the only call provided for fetching items in a transaction block. If data for only one main-chain block is wanted, an orig = 0 and a cnt = 1 are used.

Outer Module Chaining Responsibilities

For every outer call received by an outer module for which the returned status indicates incomplete status reporting (status bit 5 equal to zero), the module must arrange for adequate holding and down-chain-inclusion of all other related blocks required for future status updating.

Calls to Set Transaction Block Items

The following calls permit an outer module to set the flags, the TBE relative pointer, and the transaction status respectively in its transaction blocks.

```
call tbm$set_flags(tbindex,flags,cstatus);

call tbm$set_tbe(tbindex,tbeptr,cstatus);

call tbm$set_status(tbindex,status,cstatus);

     dcl tbindex bit (18),   /*TB index*/
      flags bit (18),    /*outer module flags*/
      tbeptr ptr,        /*TBE pointer*/
      status bit (144),  /*transaction status*/
      cstatus bit (18);  /*call status, see Table 1*/
```

flags is a bit string to be kept in the block for any use an outer module may desire. tbeptr is a pointer to the transaction block extension in the per-ioname segment; the TBM stores the corresponding relative pointer. status is the transaction status bit string. When an outer module returns to the I/O switch, the

switch always calls the TBM to store the status parameter in the corresponding block. Thus, the outer module itself need not do so; it uses the set status call only for updating old status strings.

Transaction Block Segment Switching

The following call is used by the Device Manager Process Driver and Dispatcher to switch the TBS (see Section BF.2.23 and BF.2.24).

```
call tbm$tbs(tbsp,event,cstatus);

     dcl tbsp ptr,        /*pointer to special TBS*/
      event bit (70),     /*event name for locker*/
      cstatus bit (18);   /*call status, see Table 1*/
```

The TBM normally uses the TBS created for the process group in which it is called. The tbm$tbs call causes the TBM to use the segment pointed to by tbsp as a special TBS. If tbsp is null, the TBM will revert to using the regular TBS. When operating using the special TBS, the TBM calls locker$try and provides event as the event to be signaled when the TBS is free; the TBM does not wait for the signal but returns to its caller and indicates in cstatus that the TBS was not available.

Buffer Discipline

Outer modules which need to hold user data between incoming calls must conform to a buffering discipline. It should be recalled that only DSMs overtly buffer data to implement read-ahead and write-behind and that other modules are entitled to keep only unavoidably read-ahead data (see Section BF.1.04). However certain modules may need to keep copies of processed output data as a precaution against an error occurring prior to physical completion of the output. For example, a code conversion module should keep processed output data until physical completion is indicated.

A data buffer takes the form of a transaction block extension of a block allocated into the buffer chain based in the PIB. The outer module allocates a new block by calling tbm$allocate with holdn = 2 to hold the block. When the buffer is no longer needed, the block is released by calling tbm$reset_hold with holdn = 2. The block along with the buffer (TBEs) will eventually be deallocated automatically. The declaration of the buffer TBE has the form of a standard TBE (and PIBE).

The standard buffer discipline includes the following standard down chaining. All buffer blocks whose TBEs hold data for a given incoming call are included in the down chain of the transaction block for that call. The data transmission involved in an outgoing call (to the next module) can be concerned with only a part or all of one buffer. All the blocks for outgoing

calls corresponding to a buffer are included in the down chain of that buffer block. This is shown in Figure 1, if the top, middle, and bottom rows are considered to be incoming call blocks, buffer blocks, and outgoing call blocks respectively. For example, the TBE attached to block B2 contains data provided by incoming calls corresponding to blocks A1, A2, and A3; this data was passed on with outgoing calls corresponding to blocks C3 and C4.

This chaining enables straightforward status updating by starting with the module's call chain and chasing down to the buffer chain and then down to the outgoing call blocks. DSMs engaging in read-ahead create buffer blocks prior to the corresponding read calls; under these circumstances the down chaining from the call chain is not used.

Examples of TBM Use

The examples included herein are intentionally concise to focus on the steps being demonstrated. In particular, irrelevant but usually necessary intervening code is simply omitted.

The following is an example of outer module code involved in receiving an outer call, relaying it, holding, chaining the two call blocks, and returning; minimum status handling is shown.

```
tbin = substr(in_status,127,18);
call write(pibp->pib.next_ioname,...,status1,pibp);
call hold(status1,cstatus);
tbout = substr(status1,127,18);
call tbm$thread(tbout,tbin,cstatus);
substr(in_status,1,126) = substr(status1,1,126);
return;
```

The following example shows the allocation of a buffer block, the threading of the buffer block into the call block's down chain, and the allocation of the TBE (buffer). Writing out the buffer and threading the outgoing call block into the buffer block's down chain would be similar to the previous example.

```
tbin = substr(in_status,127,18);
bcbp = addr(pibp->pib.chain_base.blindex);
call tbm$allocate(bcbp,2,tbx,cstatus);
call tbm$thread(tbx,tbin,cstatus);
/*compute any variable lengths for buftbe*/
allocate buftbe in (pibp->pib.ioarea) set (tbep);
call tbm$set_tbe(tbx,tbep,cstatus);
/*copy user's data into buffer*/
```

The next example shows a general method of updating status. The following is a declaration for two structure arrays to be used when calling tbm$get_chain.

```
dcl 1 mlist (N),
  2 (tbx,flags) bit (18),
  2 status bit (144),
  2 tberelp bit (18),
 1 list (M),
  2 (tbx,flags) bit (18),
  2 status bit (144),
  2 tberelp bit (18);
```

The first N blocks in the call chain are chased from the t1 end by issuing the following call.

```
call tbm$get_chain(pibp->pib.chain_base.t1index,3,0,N,
                   addr(mlist),cstatus1);
```

If there are less than N blocks in the chain, the redundant mlist(i) are set to zero; cstatus indicates if there are more than N blocks. To chase the down chain in the i-th block from the d1 end, the following call is issued.

```
call tbm$get_chain(mlist(i).tbx,1,1,M,addr(list),cstatus2);
```

The list(j).status are examined to update mlist(i).status. Such updating for arbitrary numbers of blocks in these chains is accomplished by using program loops to repeat the calls when more than N and/or M blocks are present respectively.

Status Returned by the TBM

Table 1 specifies the status bits returned by the TBM for all calls.

TBM Initialization

When a process group or a device manager process is created, the following call is made to the TBM to cause initialization of the TBM and TBS.

```
call tbm$init(dir_sw,cstatus);

    dcl dir_sw bit(1);  /*switch indicating in which
                          directory TBS is to be created.
                          0,1=group,process directory*/
```

During process group creation io_control$init calls tbm$init with dir_sw=0, and during device manager process creation disp$init calls tbm$init with dir_sw=1 (see Sections BF.3.01 and 2.25 respectively).

Table 1.

Status returned by the Transaction Block Maintainer.

Bit	Meaning when set to one
1	invalid TB index. (e. g. refers to vacant list)
2	caller validation level too large.
3	invalid argument. (e. g. type in tbm$get_chain not 1,2,3, or 4)
4	invalid request. (e. g. attempt to "remove" an orphan block)
5	TBS overflow.
6	invalid TBS (special TBS mode).
7-16	unassigned.
17	TBS locked (special TBS mode).
18	additional chaseable blocks in main/down chain.

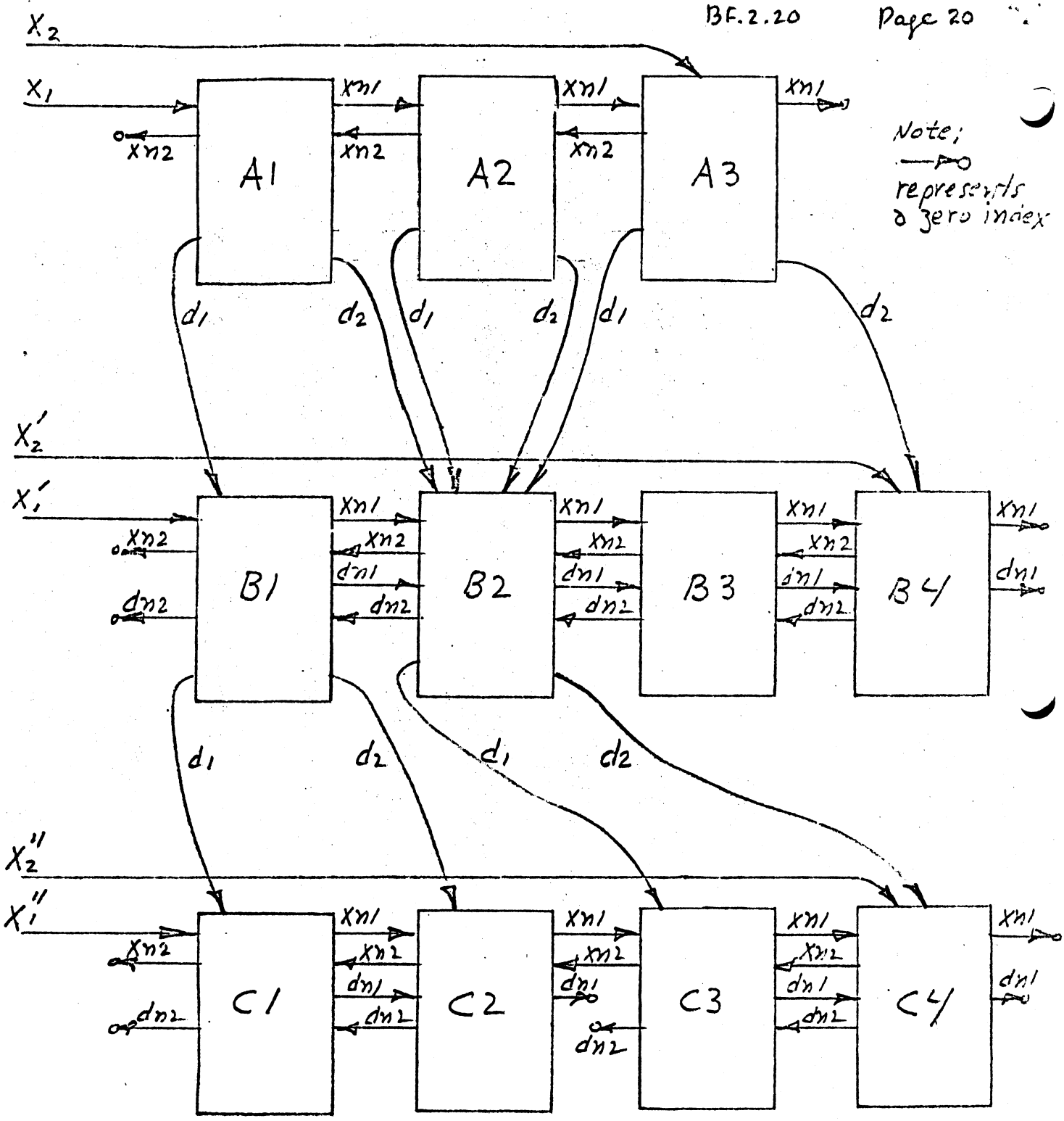

FIGURE 1. Example of transaction block chaining (See text).

BF.2.20 J.F. Ossanna 08/01/67