TO: MSPM Distribution
FROM: D. R. Widrig
SUBJECT: BF.20.11
DATE: 12/01/67

This document has been expanded slightly to indicate that a "change" structure need not actually perform any modifications.

The tally-base selection algorithm has been modified to work in a more efficient manner. It is left as an exercise to the reader to verify the algorithm's effectiveness.

The "connect$list" description has been expanded to indicate usage of the call to execute device commands through the GIOC connect channel.

A small revision was made to the "connect$list" description to reflect latest changes in the Multics Connect Processor's calling sequence.

The "cread" description was slightly expanded to indicate GIM list compression activities in certain special cases.

Published: 12/01/67
(Supersedes: BF.20.11, 07/19/67)

Identification

GIM - List Editing and Activation
D. R. Widrig and S. D. Dunten

Purpose

This section is part 3 of the complete description of the GIM; see BF.20.02.

Editing a List - change$list

In order to alter pseudo-DCW's (and DCW's, if any) the DIM must make the following call:

```
call change$list (id, idx, hi_lt_lo, lrtn, changelp
                  [changenp]);
```

where the arguments are defined as follows:

```
id bit (24)              /* ID of list to be edited */

idx fixed bin (12)       /* starting item to be edited */

ht_lt_lo bit (3)         /* status switches for edit of
                            active lists */

lrtn bit (36)            /* standard GIM error return
                            word */

(changelp,               /* points to change structures */

changenp) ptr            /* ... */
```

and the "change" structures are as defined in BF.2.03, Summary of GIM Calls and Data Bases.

The GIM entry change$list should be viewed as a dispatch program which calls editing procedures into action as they are needed. In fact, change$list does not actually edit anything; it merely calls the proper editing routines.

Change$list begins by collecting an array of pointers to the offered "change" structures. A call to a special EPLBSA procedure, getargs, returns a vector of pointers to the "change" structures and a count of the number of structures offered by the caller.

The number of structures offered is checked to make sure that at least one was offered. If none were presented, the error "badcall" is set and the GIM returns. Assuming that at least one "change" structure was offered, the caller's list ID, "id", and item index, "idx" are broken down and checked by a call to check$list. Possible errors from check$list include illegal ID, "badid", LCT not found, "lctnf", list not defined, "lndef" and illegal item index, "badcall".

If the list data is valid, the GIM checks to see if the list is currently being used by the GIOC. A call to lpw$active will indicate the channel's activity. Possible errors from lpw$active include a bad GIOC number, "badcall", or the GIOC is no longer available, "giocnf". If the list is active, a different editing technique is performed. A later section, Patching Live Lists, discusses the mechanics of altering a list being used simultaneously by the GIOC and the GIM.

For an inactive list, change$list calls the GIM list editing routine, mkpdcw, to edit the pseudo-DCWs indicated by the user. Information included in the call to mkpdcw includes which LCT is being considered, which list within the LCT to edit, where to put the edited results, how many items to edit, and pointers to the appropriate "change" structures.

Errors from the list editor include the standard check$list errors mentioned above, no Class Driving Table (CDT) has been defined, "cdtnf", illegal or unusable CDT type code, "badtyp", illegal field quantity, "illfld", illegal value quantity, "illval", list space exhausted, "tmlst", and illegal action code, "pxlat".

Further details on the list editor, mkpdcw, may be found in a later section entitled Editing GIM List Items.

Upon completion of the list editing, regardless of success or errors, change$list is finished and returns.

Editing GIM List Items - mkpdcw

As discussed in earlier sections, the major design goal of the GIOC Interface Module is the facility for interpreting DCM caller's symbolic requests in a way that is both meaningful to a GIOC and convenient for further manipulation. This section discusses the translation mechanism which converts symbolic requests into a standard item, the pseudo-DCW.

To translate one or more symbolic requests into pseudo-DCWs, the GIM makes the following internal call:

```
call mkpdcw (lctp, olstno, olstx, nlstno, nlstx,
              chs, n, mrtn)
```

where the arguments are defined as follows:

```
lctp ptr                    /* pointer to user's LCT */

(olstno,                    /* old list number */

olstx,                      /* old list starting index */

nlstno,                     /* new list number */

nlstx) fixed bin (12)       /* new list starting index */

chs (*) ptr                 /* pointers to "change"
                               structures */

n fixed                     /* number of "change" structures */

mrtn bit (36)               /* standard GIM error return
                               word */
```

Upon being called, mkpdcw immediately validates the list data and gets pointers to the indicated List Status Tables (LST) via calls to check$list. The standard check$list errors may be returned. After validating the list data, the CDT pointer, "lct.cdt", is extracted from the user's Logical Channel Table (LCT). A null pointer indicates the CDT has not been selected via the define$class call. The error "cdtnf", indicating the CDT was not found, is set for a null CDT pointer. The Class Driving Table has the following declaration:

```
/* Declarations for the Class Driving Tables */

dcl 1 cdt based (p),            /* type array for specified
                                   class */
    2 tpof(6) bit(18),          /* offsets of type info */
    2 free area((800));         /* area for type & field
                                   structures */

dcl 1 tp based(p),              /* type structure */
    2 tpval bit(84),            /* initial value for type(i) */
    2 nfld bit(24),             /* number of fields for type(i) */
    2 fldof(100) bit(18);       /* fields for type(i) */
```

```
dcl 1 fld based(p),                 /* field structure */
    2 fldact bit(3),                /* substitution code for
                                       type(i).field(j) */
    2 fldend bit(15),               /* rightmost bit of
                                       type(i).field(j) */
    2 fldmsk bit(84),               /* mask to set
                                       type(i).field(j) */
    2 nv bit(6),                    /* maximum value for
                                       type(i).field(j) */
    2 val(0:  100) bit(84);         /* values for type(i).field(j) */

dcl   cdt_max fixed bin(17)
      init(6);                      /* maximum CDT type allowable */

/* breakdown of field action codes

type 0 = illegal action
type 1 = mask-value substitution
type 2 = literal substitution
type 3 = address substitution

breakdown of type codes

type 1 = status word
type 2 = CCW
type 3 = command DCW
type 4 = transfer DCW
type 5 = literal DCW
type 6 = data transfer DCW

*/
```

The user "change" structures have the following declaration:

```
     dcl 1 change based (p),            /* user change
                                           structure */

         2 op-type fixed bin(17)        /* type of change */

         2 nchanges fixed bin(17),      /* number of changes */

         2 changes (nchanges),          /* individual changes */

           3 field fixed bin (17)       /* field to be altered */

           3 value fixed bin (24),      /* alteration value */

           3 address ptr;               /* data transmission
                                           address */
```

Note: "nchanges" may be 0 indicating no changes to be made.

For each user "change" structure, the following action occurs. The op-type is extracted and validated against the CDT. An out-of-bounds op-type or an op-type for which no matching CDT type was defined result in a bad-type error, "badtyp". If the type is valid, a pointer to the appropriate "type" substructure within the CDT is constructed.

The type code is now matched against the type code of the pseudo-DCW already in existence in the old list at the old place. A matching type is interpreted as meaning the edit is to be performed on the old pseudo-DCW and then moved to the new location. For a matching type, a temporary pseudo-DCW is initialized to equal the old pseudo-DCW under consideration. Non-matching op-types indicate the GIM is to use the initial value found within the proper "type" sub-structure of the CDT. The main edit is now started.

After extracting the field count limit, "nfields", from the proper "type" sub-structure of the CDT, the following editing is performed for each specific change request within a single "change" structure. First, the field number is extracted from the "changes" sub-structure and matched against the bounds previously extracted from the CDT. Bound violations cause the illegal field error, "illfld", to be set. The CDT is checked to insure the selected field is defined; the "illfld" error will occur if it is not. For a defined field, a pointer to the appropriate CDT "field" sub-structure is generated. The action code for this field is extracted from the "field" sub-structure. A code of 0 indicates an illegal field and a setting of the "illfld" error.

Assuming a legal action code, mkpdcw now dispatches to the proper routine to perform the edit. There are three defined action codes, 1, 2, and 3. Each kind of editing will now be discussed.

An action code of 1 indicates a mask-value edit. In this form of edit, mkpdcw uses the "value" item in the "changes" sub-structure as an index into the "value" array contained within the appropriate CDT "field" sub-structure. Thus, a "value" of 5 will cause field.value(5) in the appropriate "field" sub-structure to be used in the edit. Prior to extracting the item from the CDT, the "value" index from the "changes" sub-structure is matched against the bound on value indices, "nvalues", from the CDT "field" sub-structure. Bound violations result in the illegal value error, "illval", being set. Assuming a legal "value" index, the appropriate

value in the "field" sub-structure is selected and inserted into the temporary pseudo-DCW in the bit positions indicated by the "field_mask" in the CDT "field" sub-structure. Specific examples of mask-value substitution may be found in MSPM, BF.20.01, DCM/GIM Interface Specifications. After inserting the item into the pseudo-DCW, the mask-value edit is complete.

An action code of 2 indicates a literal substitution. In literal substitution, the "value" item itself is used in the edit. The right-most 24 bits of "value" are treated as a bit string and are inserted into the pseudo-DCW using the "field_mask" as a guide. The positioning of the field is accomplished by using the "field_end" entry of the proper CDT "field" sub-structure. Further examples of literal substitution may also be found in MSPM, BF.20.01.

An action code of 3 indicates a data-address substitution. This form of substitution is used to indicate where data handled by the GIOC and GIM is to be placed or gotten. As pointed out in earlier sections, all data transmission handled by the GIOC is to a wired-down area known to the GIM. At appropriate times, the data in the area is moved to/from the user's area. Until that time, the GIM must remember where the data is to be found. The GIM remembers the user data spaces by saving the data address in the "address space" associated with each user list. (The format and manipulation of the address space was previously described in the section entitled List Structures.) A check is made to insure that the list of newly-created pseudo-DCWs has an address space allocated. If it does not, one is now allocated. Errors in allocation indicate that too many lists have been allocated. The "tmlst" error is set for address space allocation errors. Assuming a valid address space exists, the proper entry is filled in with the segment number and offset of the user's data area. The pseudo-DCW control bit indicating a valid address supplied is set. The data address substitution is now complete.

Any other action code constitutes a general translation error and results in the error "pxlat" being set. It should be noted that an unrecognized action code implies an error in the Class Driving Table since the CDT is the source of the action code.

Having processed all "changes" sub-structures, the temporary pseudo-DCW has been completely edited and is inserted into the new slot. Both the old and new list indices are incremented and the next change structure is selected for processing.

Upon exhausting all the offered "change" structures or upon coming to the end of either list, mkpdcw is finished. It should be noted that the end of the list termination is not considered as an error. Extra "change" structures will not be processed, however.

Generation of DCWs - mkdcw

Whenever the GIM finds it necessary to generate actual DCWs so that the GIOC may begin processing some I/O requests for a user, the following internal call is made:

```
call mkdcw (lctp, lstp, mrtn);
```

where the arguments are defined as follows:

```
lctp ptr                /* pointer to proper Logical
                           Channel Table */
lstp ptr                /* pointer to List Status Table
                           to be processed */
mrtn bit (36)           /* standard GIM error return
                           word */
```

It is important to note that a call to generate actual DCWs is not under direct DIM control. The GIM reserves the right to generate and release DCW lists as conditions warrant and, in general, a DCW list will exist only when the channel is active and the GIOC is performing a service for the DIM. In these cases of channel inactivity due to the channel having terminated, the DCW space will be released. It will be shown that a policy of generating DCW space only when needed results in smoother operation of the GIM.

Before generating the actual DCWs, a preliminary check is made to insure that the indicated list is defined. An undefined list, detected by the list pointer being null, results in the error "lndef" being set and an immediate return.

If the list appears properly defined, mkdcw tests the DCW-space pointer, "lst.dcw", in the List Status Table to determine whether or not the DCW space is allocated. If no space is allocated, a call to allo$dcw is made to allocate DCW space in the wired-down segment for DCWs, dcw_seg. If the allocation was successful, the DCW-space pointer, "lst.dcw", is set to point to the area allocated for the DCWs. If the allocation was unsuccessful, the error "wrkexh" is set to indicate that workspace in dcw_seg is temporarily exhausted.

Upon receiving the "wrkexh" error, a DCM writer could safely presume that repeated calls will eventually succeed as other DCW lists belonging to other users are continually being released and the resulting space returned to the general pool of free space.

After verifying that DCW space exists within the wired-down segment, dcw_seg, a tally base is selected. The following discussion relates one of the major design aspects of the GIM, the selection of the GIOC List Pointer Word tally.

A major consideration in the Multics GIOC Interface Module is the relating of hardware events and discipline within the GIOC to the symbolic environment of the DIM writer's lists. One aspect of the relation described above is the translation of hardware items such as List Pointer Word (LPW) tally into a list ID and an item within a list. Such translation is necessary since hardware status stores preserve only a few relevant items, one being the LPW tally. In order to relate the LPW tally to a particular list and item, the following scheme is used by the GIM.

Traditional use of the LPW tally in GIOC programming requires that the tally reflect the number of DCWs to be processed. Upon processing a DCW, the GIOC decrements the tally and begins processing the next DCW. When the tally reaches zero, suitable interrupts and status stores occur and the channel ceases activity. That is, one uses the LPW tally as a <u>counter</u>.

The GIM does <u>not</u> use the LPW tally in the manner described above. Instead, the LPW tally is used as a program counter and is the key to relating hardware events to DIM writer's lists. Since the LPW tally is not used as a counter, other methods, described below, are used to indicate the extent of DCW lists to the GIOC. At issue now is the method of using the LPW tally as a tracer.

Suppose that each list item of every list for a given GIOC channel had a unique number assigned to it. Then, if the GIOC were to store the proper unique number in addition to the other items stored at status channel storage time, the GIM could easily relate which list and which item is involved in the status store in question by comparing the unique number of every item until a match of the stored unique number was made. It will now be demonstrated that, by using proper advance manipulations, the LPW tally can be made to serve as the unique number assigned to each list item.

Assume that the range of all possible LPW tallies is the closed interval I to T. When the jth DCW list of length L is generated for a given channel, the GIM desires a number N to be associated with this list such that the following statements are true:

1. N will be used as the starting LPW tally such that when item 1 of the DCW list is processed, the LPW tally is N.

2. No other DCW list for this channel has an LPW tally that falls in the range N to N-L+1, inclusive. (Recall that LPW tallies count down in the GIOC.)

If the above conditions are met, then any status channel action for list j will cause an LPW tally to be stored such that the tally stored, S, meets the following relationships:

(ending LPW tally) $N-L+1 \leq S \leq N$ (starting tally for list j)

Moreover, the number S is unique. Another way of viewing the tally selection requirements is that the GIM selects a "window" of LPW tallies for a list such that the "window" does not overlap any other "window" for any other list on the same GIOC channel.

The GIM selects a tally base, N, according to the following algorithm:

1. Set N = T; Preset top of window to largest possible value.

2. Set B = N-L; Get bottom of window -1.

3. If B is less than 0, go to error. Window ran off bottom.

4. Select list from logical Channel Table (LCT).

5. If list is the test list or not defined or no DCW's allocated, go to 12.

6. Set n = selected list's tally base; That is, get the top of this list's window.

7. Set b = n-length of selected list; That is, get the bottom of this list's window.

8. If (N <=b) or (B > =n), go to 12. Test for window overlap.

9. Set N = b; Overlap, move top of test window cown.

10. If N < = 0, then go to error. Window ran off bottom.

11. Reset to try all lists again. Go to 2.

12. If no other lists, set tally base = N and exit.

13. Go to 4.

Errors indicate an inability to allocate a window of the indicated size without overlapping a previously allocated window belonging to another list or an exhaust of the LPW tally space available. For the model B GIOC attached to the Multics configuration, the LPW tally space, T, covers the range 1 to 4095, inclusive. It is felt that 4095 DCWs active at one time represent far more than any DIM writer will ever need. However, if the above-mentioned errors occur, the "tmlst" error is set indicating too many lists are currently being used. The DCW space is released, "lst.dcw" is set to null indicating no DCW space, and return is made. Assuming no errors, mkdcw begins actual DCW production by translating the pseudo-DCWs in the offered list.

For each item in the pseudo-DCW list, the following translations and actions are performed. First, the skeleton DCW is formed by copying the first 72 bits of the related pseudo-DCW. To put it another way, the pseudo-DCW bears a suspicious resemblance to a DCW.

The pseudo-DCW control bits are checked to see if this item causes a read or a write. If it does <u>not</u>, the read/write control bits from the preceding pseudo-DCW are copied into this item. This insures that a pseudo-DCW involving data transmission carries the proper flags generated by the earlier read or write command pseudo-DCW. If the item <u>does</u> initiate reading or writing, the control flags are saved for later use. Note that the CDT alone contains the control flags. Without these flags, the GIM has no way of telling whether an operation will initiate reading or writing.

The pseudo-DCW type is now picked up and inspected. A type 0 item indicates a "hole" or unused item within a list. Since the GIOC has no null-operation facilities, the GIM performs a forward scan or preview to see if any valid items remain in the list past the hole currently encountered. Supposing that a valid item does exist later in the list a transfer DCW is constructed to point to the valid item. The transfer is constructed via a call to lpw$mktra, the GIM transfer-DCW maker. The transfer DCW is placed in the DCW slot corresponding to the hole in the pseudo-DCW list. If no valid items remain in the list, a transfer to the standard "safety" DCWs is inserted into the DCW list. It will be recalled that the "safety" DCWs are a pair of DCWs which are guaranteed to stop and terminate any GIOC data channel. The "safety" DCWs are inserted in the above case to prevent the GIOC from running off the end of the list.

Both pseudo-DCW type 1, the status request, and type 2, the CCW type, are illegal if encountered in a list of DCWs. Their occurrence causes the "dxlat" error to be set indicating a DCW translation error. In the event of this error, or any DCW translation error, the entire DCW space is released and "lst.dcw" is reset to null indicative of no DCW space currently assigned.

If the pseudo-DCW type is 4, a transfer DCW is desired. The list and item within the list which is to be transferred to are extracted from the pseudo-DCW and validated via a call to check$list. The standard check$list errors may be returned. Assuming no errors, a call to lpw$mktra will generate the necessary transfer DCW.

If the pseudo-DCW type is 6, a data transfer DCW must be constructed. First, the control bits of the pseudo-DCW used by the GIM are checked to make sure the pseudo-DCW did, in fact, indicate where that data may be found (for writes) or should be sent (for reads). If no address was supplied, the "nadd" error is set and translation halted.

Assuming a valid address, the skeleton DCW is handed to the routine patch$blen to compute the number of <u>bits</u> involved in the data transmission. This count is rounded up to the next larger <u>word</u> size and sufficient space allocated in the wired-down segment, data_seg, by calling the general GIM storage allocator, allo$data. The absolute address of the space allocated is placed in the DCW's address field. Errors in allocation cause the "wrkexh" error to be set indicating work space exhaustion.

The offset of the allocated area within "data_seg" is saved in the pseudo-DCW for later use. If the pseudo-DCW control bits indicate a write operation, the data is moved from the user's area into the wired-down area in "data_seg".

For a successful translation of the above-mentioned DCW types and for pseudo-DCW type 3, command DCW, and type 5, literal DCW, the GIM continues by translating the logical status channel pointers selected by the user into physical pointers required by the GIOC. For example, the DIM writer may indicate logical pointers, 1, 7, 6, and 3 are to be used. After mapping the pointers, physical pointers 1, 2, 3, and 4 are placed in the DCWs. The mapping information, contained in the Logical Channel Table (LCT) was included in the GIM design to allow simple re-configuration without the necessity of all DIM writers recoding certain portions of their modules after every GIOC change.

Having re-mapped the status channel pointers, the fully constructed DCW is placed in the proper slot in the DCW list. Upon completion of translation of all DCWs, an extra transfer DCW is appended to insure the GIOC going into the "safety" DCWs instead of blindly running off the end of the list. The list translation is then complete.

Generation of Transfer DCWs - lpw$mktra

During translation of pseudo-DCWs into DCWs and during editing of active lists, it is sometimes necessary to generate a transfer DCW. Many issues of security and protection center around proper transfer-DCW production with the result that it was felt a separate routine concerned only with intra-list transfers was necessary. The GIM generates a transfer DCW with the following call:

```
call lpw$mktra (lctp, lstp, idx, tdcw, lrtn)
```

where the arguments are defined as follows:

```
lctp ptr              /* pointer to user's LCT */

lst ptr               /* pointer to list to be transferred
                         to */

idx fixed bin (12)    /* item in list to be transferred to */

tdcw bit (72)         /* generated transfer DCW */

lrtn bit (36)         /* standard GIM error return status */
```

Upon receiving the above call, lpw$mktra initializes itself and prepares to generate a transfer DCW. A loop is entered and the following actions are performed.

The list pointer is checked to insure the list is defined. An undefined list cannot be transferred to. The error "bdtra" is set for undefined lists.

The index of the item to be transferred to is checked. An index which is out of the range of the target list causes the "bdtra" error to be set. Assuming the target item is within range of a defined list, its pseudo-DCW type is extracted and inspected.

A type 0 item, indicating a "hole", is allowed if and only if there exists a valid item somewhere further on in the target list. If a forward scan from a "hole" in target list detects no valid pseudo-DCW item, the "bdtra" error is set. Otherwise, the transfer is accepted and the next phase of generation begins.

A type 4 item indicates the target item is also a transfer. The list number and item index of this new transfer is extracted and validated via the check$list call. The standard check$list errors may be returned. Assuming valid list data, the loop is repeated with the new transfer item. A maximum limit on the transfer chain length, "tdepth", is set during system initialization. If the chain length is exceeded, either the chain is simply too long or the user has programmed a loop of transfer DCWs. In either case, an excessive transfer chain causes the "bdtra" error to be set.

Any other pseudo-DCW type constitutes a valid item to transfer to.

Assuming a valid item to transfer to, lpw$mktra verifies that the target list has a valid DCW list. If it does <u>not</u>, (that is, "lst.dcw" of the target list is null), a call to mkdcw will generate the target DCW list. Note that this call may be a recursive call since mkdcw may have already been calling lpw$mktra. This is the usual case upon initial activation of several user lists which are tied together by user-specified transfers.

After getting the target DCW list into proper order, the transfer DCW is constructed. The DCW type is inserted, the absolute address of the target item is inserted, and the tally field (which resets the LPW tally upon the GIOC detecting the transfer) is set to the unique tally of the target item. The resulting transfer DCW is then returned to the caller.

Patching Live Lists - patch

One of the more complicated aspects of operating the GIOC is the editing or alteration of DCW lists while the GIOC is processing the list being edited. This section discusses the strategy followed by the GIM when an active list is to be altered. Special attention will be paid to the relating of the GIOC status and the efficacy of the user's edit being performed.

Reference to the change$list call discussed earlier reveals that the list edit was handled differently if a call to lpw$active revealed the list was currently active. An active list edit is performed in four major steps:

1. Create a new "auxiliary" list via a call to change$dfl.

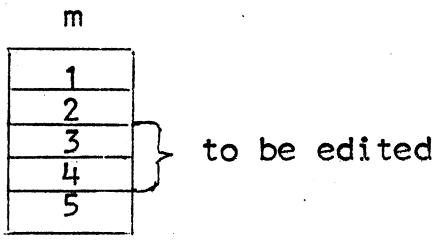
2. Perform the edit on the old list pseudo-DCWs via a call to mkpdcw.

3. Translate the new list into DCWs and patch the auxiliary list back into the main list.

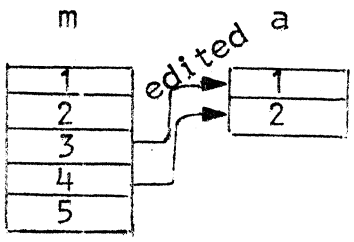
4. Compress new pseudo-DCWs and DCWs back into main list.

Consideration of the above algorithm reveals that the GIM need not be concerned with timing problems for parts 1 or 2. Part 3, however, is very sensitive as it is the phase that connects the new DCW list to the main list DCWs.

Part 3 of the above algorithm is handled by a single module and several sub-routine calls. The main patching sub-program is called "patch". For purposes of discussion, it will be convenient to refer to a diagram illustrating an actual list edit in progress. Suppose the user has indicated that list "m", of length 5, is to be edited at items 3 and 4. Schematically, the list is as indicated below:



First, an auxiliary list of length 3 is created. This list, called "a", is always created with a length equal to the span of the edit plus 1. The reason for the extra item will become apparent in a moment. A call to mkpdcw will cause list items 3 and 4 in list "m" to be edited and the resulting pseudo-DCWs placed in items 1 and 2 of list "a". Schematically, the list edit now appears as follows:



The call to patch a live list appears as follows:

```
call patch (lctp, idf, idx, aid, hi_lt_lo, prtn)
```

where the arguments are defined as follows:

```
lctp ptr                    /* pointer to proper LCT */

(idf                        /* number of main list */

idx                         /* start of patch area in
                               main list */

aid) fixed bin (12)         /* number of auxiliary list */

hi_lt_lo bit(3)             /* patch status */

prtn bit (36)               /* standard GIM error return
                               word */
```

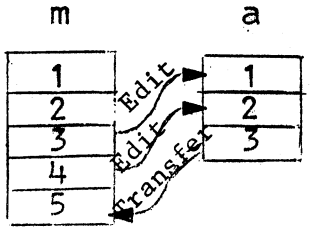
Patch is called to join the auxiliary list, "a", to the main list, "m". Patch begins by copying the main list number and beginning edit item number, ("m" and 3 in our example) into the proper auxiliary list data slots in the user's Logical Channel Table (LCT). This information will facilitate later efforts to match an auxiliary list item with its mate in the main list.

If the other auxiliary list is also being used to patch list M item 3 then a patch on patch condition exists. This condition is an error because of the difficulty the GIM would have in keeping track of the patches. The patch-on-patch error results in the "patpat" error being set.

By referring back to the Generation of DCWs section, it should be recalled that mkdcw performed the necessary steps to cause the GIOC to avoid "holes" in the user's lists by generating transfers around the "holes". If one of these "holes" occurred in the edit region in the main list, it becomes obvious that patch must undo the transfer DCWs which cause the holes to be skipped. If the transfers are not taken out or altered, then the patch will never "take" as the GIOC will always transfer over the patch area. Therefore, patch performs a backward scan from the item just before the first item to be patched, (number 2 in our example) and replaces all "hole" transfers with a new transfer that points directly at the first item in the edit area of the main list. The backward scan is stopped by reaching the top of the list or by encountering a non-"hole" item.

Having accounted for any "holes", the auxiliary list's tally base is computed. Recall from the section on Generation of DCWs that the tally base provides a means of uniquely identifying a list item. The auxiliary list tally base is set such that item 1 in the list has exactly the same tally as the first edit item in the main list. (In our example, the tally base of list "a" is set to the tally of item 3 in list "m".) This insures that the auxiliary list DCWs will appear as DCWs belonging to the main list.

After the tally base is set, a pseudo-DCW is constructed for the last item in the auxiliary list. The pseudo-DCW is a transfer-pseudo-DCW pointing at the next item past the edit area in the main list. If the next item happens to be beyond the end of the list, patch places a "hole" in the last item instead of a transfer. The lists now appear as follows:



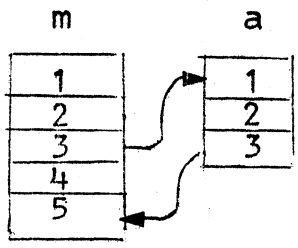
Note that the first edit item (number 3 in our example) was not altered. Thus, if the GIOC now happens to come through this area, the GIOC action is the same as it was before the alteration was performed.

A transfer-DCW is now constructed. The transfer-DCW is to be placed in the first item to be edited in the main list and will point to the first item in the auxiliary list. In other words, the transfer-DCW is the "patch" that hooks the lists together. Inspection of the section entitled Generation of Transfer DCWs reveals that if the target list has not been translated into DCWs, a call to mkdcw is made to perform the translation. Since the auxiliary list does not have any translated DCWs, the call to mkdcw is made by lpw$mktra before the transfer-DCW is generated. The standard mkdcw errors may be returned.

After getting the transfer-DCW, a pointer to the proper GIOC mailbox area is generated by a call to check$gioc. Possible errors include a bad GIOC number, "badcall", or the GIOC not available, "giocnf". The transfer-DCW is now patched into the main list DCWs via a call to patget.

Patget is a special-purpose EPLBSA routine that samples the List Pointer Word (LPW) mailbox before the patch to the DCW list is made, patches the DCW list, and re-samples the LPW mailbox. It is required that the samplings be separated by as short a time as possible. Patget will be non-interruptible during this time to insure a rapid sampling. Reference will be made later to the LPW sampling before the patch, LPW1, and the sampling after the patch, LPW2.

The lists are now joined and appear as follows:



The patching of the user's edit requests has been accomplished. Unfortunately, a formidable task remains, the determination of whether the patch "took". That is, did the GIOC arrive before, after, or during the patching operation. The next task undertaken by patch is the answering of the above question

The list data for LPW1 is found by a call to lpw$fnd. If the LPW cannct be matched against any list, lpw$fnd will return a list number of zero. Patch will assume that a list number of zero indicates the GIOC has gone into the "safety" DCWs discussed in GIOC Channel Activity. A zero list number will cause patch to set the patch status word, hi_lt_lo, to "010"b indicating the patch came too late. Similar action occurs for LPW2, the LPW sampled after the patch. Note that the assumptions for LPW2 require that the time of the sampling interval in patget is quite small with respect to the speed of the GIOC. In fact, the GIM assumes that patget performs its operations in a time that allows no more than 2 or 3 GIOC actions on a given channel.

If one assumes an appropriately brief interval between LPW samples in patget, it follows that a simple test of the item pointed at by LPW1 will reveal whether the GIOC was above or below the patch area during the patch. If the GIOC was below the patch area, the hi_lt_lo bits are set to "001"b. If the GIOC was above the patch area, the hi_lt_lo bits are set to "100"b. Of course, the terms "above" and "below" refer to higher or lower item indices in the main list. If the GIOC was working on the first item in the patch area just as the list was being patched, the patch occurred too late and hi_lt_lo is set tp "010"b to indicate this.

If none of the above tests revealed anything, patch prepares to "walk" down the list starting at LPW1. By calling a special routine named cread$step, patch takes one "step" at a time down the list in the same manner as the GIOC did. This stepping procedure continues until either the end of the GIOC DCW list is reached or the initial patch point in the main list is reached or the point indicated by LPW2 is encountered.

If the LPW2 point is reached, the GIOC was not in the patch area. In this case, the patch was effective. Patch tidies up and returns. Similarly, encountering the end of the list is interpreted as an effective patch.

If the patch point is reached, the GIM must determine which way the GIOC went. That is, there is no a priori way of telling whether the GIOC travelled down the patched fork in the DCW list or down the old, un-edited, fork. Assuming that only a small amount of GIOC action occurred between the LPW samplings in patget, the following method can be used to determine which fork was used by the GIOC.

Patch assumes that the patch "took" and the GIOC went down the patched DCW list. Repeated calls to cread$step will "walk" patch down the trail. The trace is stopped after some small number of steps (currently, 10) or when the point indicated by LPW2 is reached or when the end of the list is reached. Patch then backs up to the patch point and travels down the other fork, the old and un-edited one, until it has traveled as far (used the same number of steps) as the previous walk or until it encounters the LPW2 point. If the LPW2 point is encountered, the patch did not take so the hi_lt_lo switch is set to "010"b to indicate the patch occurred too late.

If the trail goes as far as the other fork without encountering the LPW2 point, the GIM presumes the patch "took" and the GIOC processed some or all of the new patch.

The astute reader of the above section will undoubtedly be able to convince himself of flaws in the above algorithms. On the other hand, the prototype GIOC does not readily lend itself to trail following of the type described above. It is expected that the model B GIOC will provide the GIM with at least one very powerful tool to facilitate trail-following; this tool is the ability to flag transfer-DCWs so as to cause an interrupt and a selectable "tag" stored in the hardware status word.

At the time of the installation of the model B GIOC, it is anticipated that Multics personnel will review the trail-following mechanisms contained in the patch module and implement a more sophisticated and foolproof scheme.

List or Channel Activation - connect$list

In order to activate a list or restart a channel, one requires that all symbolic editing and creation of the pseudo-DCWs be translated into actual GIOC DCWs. Also, one requires that the appropriate mailboxes and connect instructions be issued so as to cause the GIOC to commence processing the DCWs associated with the device under consideration. In order to set the electrons in motion, the DIM makes the following call to the GIM:

```
call connect$list (id, idx, lrtn [, tid, tidx]);
```

where the arguments are declared as follows:

```
(id,                        /* id of list containing CCW */
tid) bit (24)               /* id of list to start GIOC at */
```

```
     (idx,                          /* list item number of CCW to use */
     tidx) fixed bin (12)           /* list item to start GIOC at */
     lrtn bit (36)  /* standard error return word */
```

Prior to beginning actual list processing, connect$list calls out to getargs to collect pointers to the optional argument pair, "tid" and "tidx". If the number of optional arguments is neither 0 or 2, the error "badcall" is returned.

Having collected the optional arguments (if any), the list id and item index are verified through a call to check$list. Possible errors include "badid", LCT not found, "lctnf", list not defined, "lndef", and illegal item index, "badcall". A check is then made that the item in the indicated list is, in fact, a Channel Command Word (CCW). If the pseudo-DCW type is not 2, indicating a CCW, the error "notccw" is set. Assuming that a CCW is being used, the user's skeleton CCW is fleshed out by inserting twice the GIM channel number for this device. (Recall that "lct.phychn" contains half the actual physical channel number.)

If no optional arguments were present, the GIM assumes that the CCW is to be executed as a GIOC connect channel command. Such commands do not require any data mailbox preparation. Examples of connect channel commands include a "request status" command, a "rewind" command, etc.

Assuming no optional arguments, the GIM attempts to insert and activate the CCW in the manner described later in this section.

If the optional arguments were present, the GIM assumes that the caller wishes to start the channel at the indicated location. If the channel is already active, then the GIM further assumes the user has made an error. A call to lpw$active will resolve the channel's status.

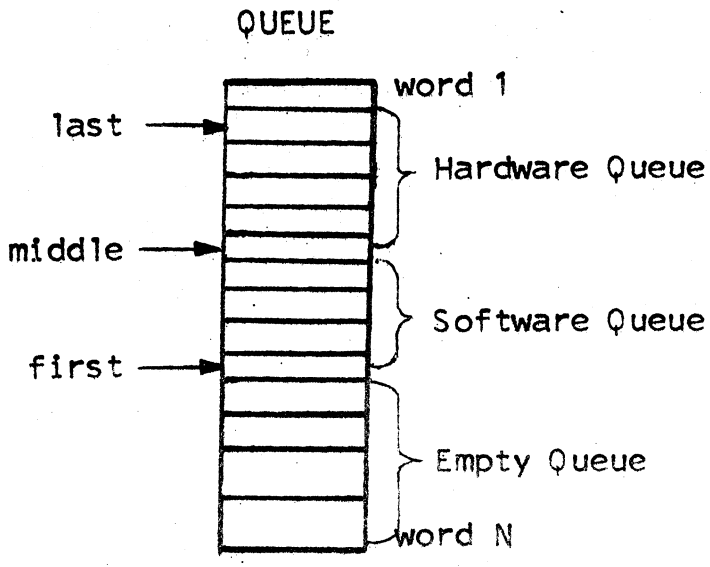
Lpw$active may report an error; either the error "giocnf" is set, indicating the GIOC is no longer usable or "badcall" was set indicating an illegal GIOC number. Otherwise, if the channel is active, the error "lpwact" is set indicating the list is active and cannot be switched.

Assuming that all is well, connect$list calls check$list again, this time verifying the list and item indicated in the optional arguments. The same errors are possible as in the earlier call to check$list.

If the new list proves to be valid, a call to lpw$set is made to set the LPW mailbox for this channel to point to the indicated starting point. It is at this point that DCWs are formed, lists are tied together, and general hardware-oriented tasks are performed. Possible error returns from the call to lpw$set include: the standard check$list returns, bad transfers within a list, "bdtra", wired-down workspaces temporarily exhausted "wrkexh", too many list items "tmlst", "status" or "connect" or unrecognizable "op-type" in list, "dxlat", no address given for data transmission "nadd", or list not defined, "lndef".

Having survived the call to lpw$set, connect$list housekeeps the Logical Channel Table for possible patching and editing of the lists while they are active.

The user's CCW is now added to the CCW queue for the indicated GIOC connect channel. The CCW queue can be considered to be a circular queue composed of 3 regions. One region, the hardware queue, contains the CCWs currently being processed by the GIOC. The hardware queue may be empty if a particular connect channel has no work to do. The second region, the software queue, contains CCWs added since the connect channel was started on processing of the hardware queue. These CCWs are currently unknown to the GIOC. An empty software queue indicates either an inactive connect channel and no work to do or no work to do after the current connect channel activity terminates. The final queue represents the unused or available queue area. At a given moment in time, the queue for a given connect channel will appear as follows:



The three pointers, "first", "middle", and "last", delimit the various sub-queues. The pointers and the queue for a given connect channel are placed in a structure allocated in the wired-down segment, dcw_seg. The structure is, in effect, a Logical Channel Table (LCT) for a GIOC connect channel and is known as a Connect Logical Channel Table (CLCT). It has the following per-channel declaration:

```
/* Declarations for connect channel buffering and control */

dcl 1 clct based(p),                          /* connect channel's
                                                 LCT */
    2 lock bit(36),                           /* interlock */
    2 actsw bit(1),                           /* ON if active */
    2 fst fixed bin(17),                      /* index of first CCW */
    2 mid fixed bin(17),                      /* mid-point index */
    2 lst fixed bin(17),                      /* index of last CCW */
    2 ccwbuf(10 /* cbufsiz */) bit(36);       /* buffer for CCWs */

dcl   cbufsiz fixed bin(17) ext static;       /* length of CCW buffer */
```

A pointer to the selected CLCT is obtained by connect$list via a call to check$connect. Errors returned may include illegal GIOC or connect channel number, "badcall", GIOC not usable, "giocnf", or connect channel not usable, "connf". Check$connect generates the pointer to the proper connect channel by inspecting information (relating the status of each CLCT) found in the Channel Assignment Table (CAT).

Having gotten the proper CLCT, the caller's CCW is added to the end of the software queue. The "first" pointer is then incremented to reflect the addition. If the "first" pointer has run off the end of the queue, it is reset to the beginning of the queue. The resetting of the pointer effectively causes the queue to be treated as a circular queue.

After updating the "first" pointer, it is tested against the "last" pointer. If they are equal, the empty queue has been completely exhausted and no more requests can be accepted on this connect channel. Connect$list will rest in a busy loop until the queues are reduced by GIOC activity.

Assuming the queues are in proper order, a test of the "active" switch for this connect channel is made. If the channel is active, no more action need be taken as the GIOC will eventually get to the CCW just added to the channel's queue. If the channel is inactive, it must be started.

To activate the proper connect channel, a proper Command Pointer Word (CPW) must be placed in the connect channel's mailbox. Also, it is presumed that the entire hardware queue, if any, has been processed and can be scrapped. This presumption hinges on a GIOC connect channel terminating for only one reason, completion of processing of all CCWs given to it on a connect call.

The CPW is initialized by setting a status channel pointer for the "exhaust" condition. This insures the GIM being informed when the channel finishes processing its list of CCWs.

The hardware queue is scrapped (and the empty queue expanded) by moving the "last" pointer up to the "middle" pointer. The "middle" pointer is moved up to the "first" pointer; unless "first" is lower than "middle", indicating a circular queue, the extent of the queue is determined by the upper end. If the "first" pointer has wrapped around to the bottom of the queue, "middle" is set to point at the upper end of the queue. Consideration of the above algorithm reveals that a circular software queue is effectively broken into two queues, a hardware queue and another (smaller) software queue.

The number of CCWs to be processed by the GIOC is simply the difference between the new "middle" pointer and the new "last" pointer, the delimiters of the hardware queue. This difference is placed in the CPW tally field and the absolute address of the CCW pointed at by "last" is placed in the CPW address field.

A Connect Operand Word (COW) is formed using the symbolic connect number for this connect channel. The COW is formed in accordance with requirements set forth in MSPM, BK.5.01, the Multics Connect Processor.

The "active" switch in the CLCT is set ON. The CPW is inserted in the proper mailbox and a call to the Multics Connect Processor is made.

Copying Data Into a ser's Area - cread

One of the most delicate issues encountered when dealing with an asynchronous I/O device such as the GIOC is the determination of when the data associated with a particular DCW list has been processed. In general, the GIM is confronted with three major problems:

1. When to release areas dedicated for output
2. When to consider input as completed
3. When to compress DCW patches into the patched or main list

The GIM resolves the above-mentioned problems by a single call of the following form:

```
call cread (lctp, crtn)
```

where the arguments are declared as follows:

```
lctp ptr        /* pointer to a Logical Channel Table */
crtn bit (36)   /* standard error return word */
```

The cread module is charged with determining GIOC activities and how they relate to DIM caller's data areas. In effect, one could view cread as a module that follows the GIOC around and tidies up data transmissions, updates list entries, etc.

Initial setup includes getting the GIOC number and GIM channel number for the indicated LCT. The GIOC number is used to get a pointer to the GIOC base via a call to check$gioc. Errors returned include illegal GIOC number, "bdcall", and GIOC not available, "giocnf". Assuming all is in order, the current LPW and DCW mailboxes for the channel indicated by the LCT are extracted and examined. A call to lpw$fnd will relate the LPW mailbox to a particular list and item within the list. Possible errors include only a system or machine error, "syserr". Reference to the entries "lct.copid" and "lct.copidx" will reveal the last known starting place of all unexamined list activity. It will be the task of the cread module to start at this location and "walk" along the DCW lists until it reaches the current list and item index. The following discussion describes various bookkeeping functions performed on the journey.

A particular list and item within the list is selected for consideration. The pseudo-DCW is examined to see if it is a data transfer DCW and if the transmission was directed from an external device to core memory. That is, a test of a data transfer DCW is made to see if reading was involved. For reading, the following actions occur for each item selected.

A call to patch$blen will return the total number of bits to be transmitted by the DCW in question. If any of these bits have already been copied by earlier calls to cread, the entry "lct.copbtc" will contain the count of those bits already copied. A test is now made to determine whether the item under inspection and the current GIOC DCW are one and the same. If they are the same one, then only that data already input by the GIOC can be safely copied. A call to patch$blen will return the total amount of bits left to be transmitted by the GIOC DCW; the determination of how many have been transmitted is then quite straightforward. For coincident DCWs, a flag is set to indicate that cread can go no further. If the items are not coincident, all of the bits not copied by an earlier call can be copied. In addition, the flag is reset to indicate cread may continue processing.

Having deduced how many bits of the data area are to be copied, the GIM references the "address space" vector to determine where to place the data. A simple move then places the data in the user's area.

After moving the data, a call to cread$step advances the trail following mechanism by one GIOC "step". If the new advancement reaches the logical end, not physical end, of the lists, the scan down the lists terminates; a call to lpw$safe guarantees that the channel has stopped and will remain stopped. (See GIOC Channel Activity for a discussion of active channels.) Possible errors from lpw$safe include bad GIOC number, "badcall", and GIOC not available, "giocnf". After making sure the channel is stopped, all DCW lists and data areas for the proper channel are released via a call to mkdcw$free.

After processing all relevant items, cread updates the information relevant to the last item processed, "lct.copid", "lct.copidx", and "lct.copbtc". Attention is then turned to the compacting of patches back into lists being patched. (See Patching Live Lists for a discussion of patching.)

This is done in 2 steps. The old DCW's in the main list are replaced by the new DCW's in the auxiliary list. The auxiliary list is then released. The first step may be performed anytime after any read buffers associated with an old DCW have been copied to the user's area and the GIOC is not working on an old DCW. The data areas associated with the old DCW's are freed and the old DCW's replaced by the new. This is done from the bottom of the list to the top so that the transfer to the auxiliary list is replaced last.

Once the transfer is done the GIOC enters the auxiliary list. If the auxiliary is not being used now by the GIOC or the read-copying program it never will be and can be released. The data areas are not released as they are now being used by the main list. Everything is now back to normal and the patch is forgotten.

If the auxiliary list is being used by the GIOC, the list compression is skipped.

Note that another compression attempt may take place at a later time.