TO: MSPM Distribution
FROM: R. R. Widrig
SUBJECT: BF.20.13
DATE: 12/01/67

The document has been expanded to include a description of a new entry, "check$device_name".

Published: 12/01/67
(Supersedes: BF.20.13 07/19/67)

Identification

GIM - Miscellaneous
D. R. Widrig and S. D. Dunten

Purpose

This section is part 5 of the complete description of the GIM: see BF.20.02.

General List Utilities-check$list, check$device_index, check$gioc, check$connect, check$statusp, check$device_name

Many of the GIM procedures require validation and/or generation of data relevant to a particular device. For instance, the GIM may need a pointer to a user's Logical Channel Table (LCT), or the GIM may wish to verify that an item index is contained with a list, etc. The various checking and generation routines are contained within a single module named "check". The various routines are described in the following section.

A moments inspection reveals that many of the items relevant to a user's list are quite inter-related. Specific relations may be found among the following list items:

1. list id
2. Logical Channel Table (LCT)
3. list number
4. item index
5. List Status Table (LST)

To derive and/or check the validity of the above-mentioned items, the GIM makes the following call:

```
     call check$list (control_bits, id, lctp, idf, idx,
                      lstp, lrtn)
```

where the arguments are declared as follows:

```
     control_bits bit(8)      /* check and verification control */
     id bit(24)               /* list ID */
     lctp ptr                 /* pointer to user's LCT */
     (idf                     /* list number */
     idx) fixed bin(12)       /* item index */
     lstp ptr                 /* pointer to LST */
     lrtn bit(36)             /* standard GIM error return word */
```

The variable "control-bits" is used to control the checking and validation of the list data. It can be conceived of as a micro-coded dispatch table with the following meaning:

Bit Number	Meaning if Bit is 1
1	Derive LCT pointer from ID
2	Derive list number from ID
3	Not used
4	Derive LST pointer
5	Check LCT pointer
6	Check list number
7	Check item index
8	Check LST pointer

Consideration of the items involved quickly reveal that many subtle inter-relationships exist. For instance, a request to derive an LST pointer requires prior validation of the list number and the LCT pointer as these two items are necessary for deriving a LST pointer.

Assuming that various consistency inter-relationships of the type mentioned above are handled automatically by the check$list procedure, the following items are (or can be) tested:

1. LCT pointer validity

   Errors include: illegal logical channel number in id, "badid". LCT not found, "lctnf".

2. List number validity

   Errors include: illegal list number from bad id: "badid".

3. LST pointer validity

   Errors include: list not defined: "lndef".

4. Item index validity

   Errors include: bad item index: "badcall".

The device name offered by a DIM caller as a result of its receiving an "attach" call can be checked and processed by an inter_GIM call of the form:

```
     call check$device_name(device_name,dct_index,device_index,
          drtn)
```

where the arguments are defined as follows:

```
     device_name char(*)       /* name of device in DCT */
     dct_index fixed bin(17)   /* returned index of device in
                                  DCT */
     device_index fixed bin(17) /* device index from DCT */
     drtn bit(36)              /* standard GIM error return
                                  word */
```

Check$device_name scans every entry in the Device Configuration Table searching for a match of "device_name". Upon finding a match, the entry number of the matching name is returned as "dct_index". The "device_index is returned from the data found in the matching entry.

Errors returned include only a name for which no match can be found, "badcall".

The device index offered by a DIM caller in such calls as request$status and define$list can be verified by a call of the form:

```
     call check$device_index (device_index, lgch, lctp, drtn)
```

where the arguments are defined as follows:

```
     device_index fixed bin(17)       /* user device tag */
     lgch fixed bin(12)               /* logical channel number */
     lctp ptr                         /* pointer to LCT */
     drtn bit(36)                     /* standard GIM error
                                         word */
```

Check$device_index calls out to the inter-process communication ackage (See BQ.6.01) to get the relationship between the device index, "device_index", and the logical channel number, "lgch". The logical channel number is returned to the caller. The logical channel number is then verified to insure that it is within the proper bounds. An error results in the "baddev" error. Assuming the logical channel number is within the proper bounds, the proper LCT segment number is extracted from the Channel Assignment Table (CAT) and checked. A segment number of zero indicates no LCT is currently defined for this logical channel. This error causes the "lctnf" error to be set. Assuming a legal segment number, a pointer to the LCT is constructed and check$device_index returns triumphant.

Several other utility routines included in the check module are:

```
     check$gioc (giocno, gioc_ptr, grtn)
     check$connect (giocno, connect_no, connect_ptr,
                    gioc_ptr, crtn)
     check$statusp (giocno, statno, status_ptr, gioc_ptr, srtn)
```

where the arguments are declared as follows:

```
     (giocno                    /* GIOC number */
     connect_no                 /* connect channel number */
     statno) fixed bin(17)      /* status channel number */
     (gioc_ptr                  /* pointer to GIOC base */
     connect_ptr                /* pointer to connect channel
                                   LCT */
     status_ptr) ptr            /* pointer to status channel LCT */
     (grtn                      /* standard GIM error return
                                   word */
     crtn                                       ...
     srtn) bit(36)                              ...
```

All of these routines validate the input arguments and return the proper pointer to the desired data base. The information relevant to each data base is contained within the CAT and is processed in a manner similar to the processing of the LCT pointer in the check$device_index call.

Setting an LPW Mailbox- lpw$set

The GIM makes the following call when it is desired to set the list channel mailbox:

```
    call lpw$set (lctp, lstp, idx, srtn)
```

where the arguments are defined as follows:

```
     lctp ptr                   /* pointer to user's LCT */
     lstp ptr                   /* pointer to list to point LPW
                                   to */
     idx fixed bin(12)          /* index of item to point LPW to */
     srtn bit(36)               /* standard GIM error return word */
```

Upon receiving this call, lpw$set calls lpw$mktra to make a transfer DCW which points at the proper item in the indicated list. Inspection of lpw$mktra reveals that lists with no currently defined DCWs are translated and readied for use. Having gotten the transfer DCW from lpw$mktra, one makes the shrewd observation that the only difference between a transfer DCW and an equivalent LPW mailbox is the 3-bit type code. Thus, lpw$set transforms the transfer DCW into an LPW mailbox entry by simply resetting the DCW type code.

A call to check$gioc will verify that a working GIOC is to be used and will return a pointer to the GIOC mailbox area. Errors include an unusable GIOC, "giocnf" or a bad GIOC number, "badcall".

Assuming no errors, the LPW is placed in the proper mailbox via a call to double$store. Double$store is a tiny, machine-coded, routine which accomplishes the setting of the 2-word mailboxes by such double-word operations as STAQ. This is necessary since setting only one word of the mailbox at a time could run into embarrassing and unpredictable GIOC behavior.

Having inserted the LPW into the mailbox, a copy is placed in the user's LCT at the entry "lct.stlpw" for later use in the lpw$fnd call. Lpw$set then returns.

<u>Relating a LPW to a List</u>- lpw$fnd

At certain times during editing of active lists and during the request$status call from a DCM writer, the GIM needs to be able to relate a hardware List Pointer Word (LPW) mailbox contents to a particular list and item within the list. To relate the above quantities, the GIM makes the following call:

```
    call lpw$fnd (lctp, fbit, fidf, fidx, flpw, rtnf)
```

where the arguments are declared as follows:

```
    lctp ptf                  /* pointer to user's LCT */
    fbit bit(1)               /* ON if LPW has not moved since
                                 startup */
    fidf fixed bin(12)        /* list number of related list */
    fidx fixed bin(12)        /* index of related item */
    flpw bit(72)              /* test LPW to be related */
    rtnf bit(36)              /* standard GIM error return
                                 word */
```

Upon receiving the call, lpw$fnd starts by setting the list number, "fidf", and the item index, "fidx" to 0 indicating no related list or item could be found. The address field contents within the LPW are extracted for later use. The LPW is then matched against a copy of the starting LPW which was saved during the last time the list was activated. (Recall that this item was saved in the user's LCT as "lct.stlpw" during the lpw$set call from connect$list.)

A match indicates that the user's LPW is still pointing at the first item and has not moved. The caller of lpw$fnd may be interested in knowing this so "fbit" is set ON to indicate it.

A mis-match indicates the LPW has moved since the list was activated. To put it another way, the GIOC has done some processing on the DCW lists. A mis-match causes "fbit" to be set OFF and the absolute address saved earlier to be backed up 2 locations. This "backing up" or decrementing of the LPW address reflects the fact that the GIOC LPW discipline is such that the LPW always points to the _next_ thing to be done. That is, the item of interest is the one immediately _before_ the LPW address.

Having gotten the LPW address, a search is made of all defined lists which have defined DCW lists. Errors in conversion of pointers into absolute addresses will cause the system or machine error, "syserr", to be set. For each DCW list, the span of absolute addresses covered by the list is checked to see if it covers the LPW address. If it does, the list related to the offered LPW has been found. Simple arithmetic will get the item, lpw$fnd returns.

If no list spans the LPW address, the default settings for the list number and item index are returned. This case is not considered an error.