TO: MSPM Distribution
FROM: D. R. Widrig
SUBJECT: BF.20.01
DATE: 12/01/67

Slight modifications have been made in the descriptions of the "define$channel" call to reflect the latest GIM changes. Also, the "define$release" call has been clarified slightly.

Published: 12/01/67
(Supersedes: BF.20.01, 07/11/67;
BF.20.01, 05/10/67;
BF.20.01, 12/15/66)

Identification

DCM/GIM Interface Specifications
H. S. Magnuski and D. R. Widrig

Purpose

This document and the MSPM section on the internal structure of the GIM (BF.20.02) describe the hardcore modules which have direct control over all of the input and output operations in Multics. The input to the GIM is a sequence of calls from a Device Control Module (DCM) which defines the strategy for running a particular device in the system. The output of the GIM is a list of instructions issued to the GIOC adapter which carries out the actions specified by the input calls. This document first explains how a strategy (example: first activate the channel and turn on the request-to-send lead. Send two synch characters followed by twenty characters of data from buffer alpha.) can be converted into a form understandable and acceptable to to the GIM. It then explains how to use the calls to the GIM to translate this strategy into GIOC instructions and feed these instructions to the GIOC adapters.

Introduction

In designing a strategy for a device the DCM writer has a choice of no more than five different methods or types of control. The types of control are

a) Channel Command Words, used primarily for activating or terminating action in a GIOC channel.

b) Command Data Control Words (DCW's) for setting up a GIOC channel.

c) Transfer DCW, an unconditional jump instruction for the GIOC.

d) Literal DCW, for sending a stream of constant characters to the device.

e) Data DCW, for transferring data to and from the external world.

In addition to the five controls above, there is one type of return information available to the DCM

f) Interrupt Return Status.

Items a to f constitue six types (hereafter labeled "op_type") of control information which can be passed back and forth between the device and the device manager, and these six op_types are the only means available to the DCM to control its device.

The Class Driving Table

It was mentioned above that all six op_types may not be available to a particular DCM. The restrictions on op_types and much other vital information is contained in a set of tables known as the "Class Driving Tables" (CDT's). The CDT's are by far the most valuable and sensitive tables used by the GIM, for they contain both the information needed by the DCM to run its device, and the restrictions imposed on that DCM to insure the security of the I/O system.

In order to perform any I/O operation the DCM must first have access to the Class Driving Table suitable for his device. The DCM never accesses the CDT directly, only through calls to the GIM. Thus, the access right to the table implies the permission to use the class of instructions contained within the CDT. The access rights to the table are contained within the file system access control mechanism and thus the security of the I/O system depends primarily on the proper use of these access rights.

Each Class Driving Table for the GIM is in reality an array of structures with the following declarations:

```
dcl  1 cdt based(cdt_ptr),
       2 type_offset(6)bit(18),     /* offset into type info  */
       2 free area((1024));         /* area for sub-structure */

dcl  1 type based(type_ptr),
       2 type_mask bit(84),         /* initial value for type */
       2 nfields bit(24),           /* number of field items  */
       2 field_offset(100)bit(18); /* offset into field info */

dcl  1 field based(field_ptr),
       2 field_action bit(3),       /* action code            */
       2 field_end bit(15),         /* shift factor for action
                                       code                   */
       2 field_mask bit(84),        /* field definition mask  */
       2 nvalues bit(6),            /* number of values for
                                       field                  */
       2 value(0:nvalues)bit(84);  /* substitution values    */
```

Each of the six major structures for the CDT array corresponds uniquely to the six op-types mentioned above:

```
cdt (1) = interrupt status return
cdt (2) = channel command word
cdt (3) = command DCW
cdt (4) = transfer DCW
cdt (5) = literal DCW
cdt (6) = data DCW
```

The substructures within each array have their properties explained in the following paragraphs.

The eighty-four bit string "type_mask" initializes the pseudo-word being generated during the first time one of the op-type substructures is called into action. These pseudo-words which are generated combine into either pseudo-DCW-lists or pseudo-CCW-lists, and these pseudo-lists are eventually transformed into the series of DCW's which control the I/O activity of the GIOC.

"Type_mask" is used to indicate which bits are to be initialized, to be zero or one. Seventy-two of the eighty-four bits generate the pseudo_DCW; six bits are for use during the global change call, which is described later. Another bit indicates that the DCW will be used for reading, another indicates writing, and a third is used by the GIM to control certain read-write operations. Of the remaining three bits, one is used by the disc DCM only, the second indicates that this DCW will terminate activity on the channel, and the third is reserved for use by the GIM.

The second level element "nfields" gives the size of the "field" substructure array of the CDT. The concept of a "field" in the pseudo-word is an important one, for the editing of the contents in the fields of a pseudo-word is the only means of creating specialized I/O instructions for the adapters. Each element of the field substructure array defines a field in a particular op_type, and the only fields which can be altered by the DCM are the ones specified within the field substructure array.

The manner in which a given field can be altered is indicated by the value of the element called "field_action". There are three possible kinds of actions which may be used to change the contents of a field:

1. mask-value substitution
2. literal substitution
3. data address substitution

Only one type of field_action can be specified for a given field.

In mask-value substitution the bits to be changed are chosen by "field_mask", and the value of the bits is given by the element "value (i)" in the value array. In this case the concept of a field is extended to include a "pattern" of bits sprinkled throughout the pseudo-word, not just a contiguous block of bits starting at bit n and ending at bit n + x. Figure 1 shows how an 18 bit pseudo-word might get modified after mask-value substitution. The field and value numbers used in the examples are fictitious and are used for demonstration only.

Figure 1A shows the initialized pseudo-word for a particular op_type. Figure 1B is the mask used for field 1 and Figure 1C is value number 3 of field 1 for this op_type. Figure 1D is the result of mask-value substitution on Figure 1A.

In literal substitution, the right most bit position of the field in the pseudo-word is specified by "field_end", and the effective size of the field is determined by "field_mask". The literal to be substituted into this field is handed to the GIM by the DCM in the "changes" structure which will be specified later in this paper.

Figure 1F shows the result of a literal substitution of the literal "011100"b performed on Figure 1D.

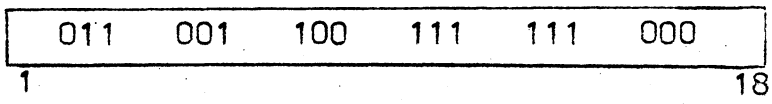
In data address substitution the address of the data is handed to the GIM by the DCM in the "changes" structure which will be specified later.

When the substitutions to the fields in the pseudo-word have been completed, the pseudo-word is combined with the four bits of op_type to become an element in a pseudo-list. Each pseudo-list is an array of structures with the following declaration:

```
dcl 1 pseudo_list (size) based (p),  /* size is the length
      2 op_type bit (4),                of the list */
      2 pseudo_word bit (84);
```

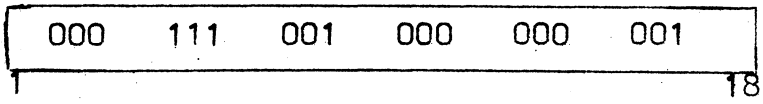
Fig. 1 - Generation of 18 bit pseudo-word through field substitution

A.

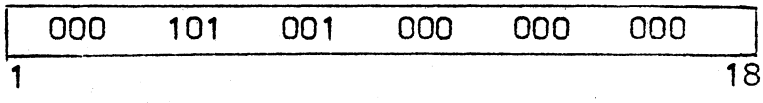


a.) Initialized pseudoword after type_mask, operation

B.

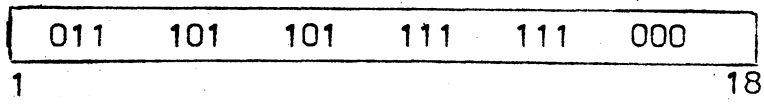


C.



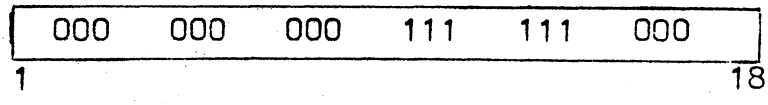
b.) Definition of field 1 and c.) value 3 of field 1.

D.



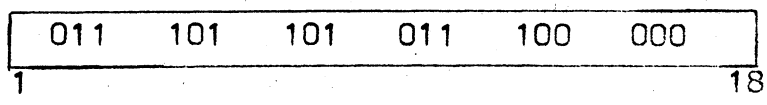
d.) Pseudo word after mask-value substitution

E.



e.) "field_mask" for field 2. "field_end" = 15

F.



f.) Pseudo word after literal substitution with value = "011100"b

If any pseudo-word in the pseudo-list is a data DCW, then there will also be created an address list which is in one-to-one correspondence with the pseudo-list that created it. The declaration of the address list is

```
dcl 1 addr_list (size) based (p), /* optional data address
                                     array */
    2 segno bit (18),  /* segment number from addr ptr */
    2 offset bit (18); /* segment offset from addr ptr */
```

The one op-type which is not used to generate a pseudo-word is "interrupt status return". In this case the Class Driving Table is used to interpret an incoming status word. The interpretation of this word is outlined when the GIM "request$status" call is specified.

In summary, the DCM can translate its device strategy into suitable instructions by specifying an op_type, the fields within that op_type, and the contents within the fields. The allowable op_types, fields and contents for a particular device manager are contained within the Class Driving Table to which that DCM has access. The generation of the Class Driving Tables and their contents are described in Section BF.20.06. The pseudo-words which are generated through use of the CDT's are formed into pseudo-lists, and these pseudo-lists are again transformed into the real lists of instructions which control the input and output devices in Multics.

The next section of this paper describes how these pseudo-lists can be created and destroyed, and how these lists can be used to control the activities of the peripheral equipment.

Creation and Control of Pseudo-Lists

In order for any pseudo-list to be created the user must first gain control of the logical channel he wishes to operate, and then his DCM must have access to a CDT table to create the pseudo-words to be used in controlling the channel.

Obtaining control over a logical channel is not done with a call to the GIM, but is done with the help of a third party such as the Answering Service or Transactor (BT.1.02). When the Transactor decides it can allow the DCM to use a particular channel it makes the following privileged call to the GIM:

```
define$channel (device_name, device index, event_id, rtn_stat)

dcl device_name char(*),          /* device to be used */
    device_index fixed bin (17), /* user device tag  */
    event_id          bit (70), /* event channel
                                   identification
    rtn_stat          bit (36);
```

When the DCM has gained control over a particular logical channel, the next step needed to create a pseudo_list is to define a class of op_types to be used with a logical channel. While trying to define this class of op_types the DCM's access rights to the corresponding CDT are checked, and if the call is valid, then the DCM can begin to create a new pseudo_list. The call to define a class of op_types looks like:

```
define$class (device_index, class_id, rtn_stat)

dcl device_index fixed bin (17), /* index of device in DSTM
                                    tables                    */
    class_id char (*),           /* class identifier          */
    rtn_stat bit (36);           /* return status (from GIM) */
```

The argument "class_id" specifies the name of a segment known to the GIM which contains the CDT information.

The meaning of the bits in rtn_stat will be specified in the summary of GIM calls and data bases, BF.20.03, and BF.20.05.

Assignment of Space for the Pseudo-List

Once the DCM has defined a class of operations, he must then tell the GIM that he wants to create a new pseudo-list. This is done by issuing the following call:

```
define$list (id, device_index, lgth, rtn_stat)

dcl id bit (24),                 /* list identification (from
                                    GIM)                      */
    device_index fixed bin(17), /* user device tag            */
    lgth fixed bin (12),         /* list length               */
    rtn_stat bit (36);           /* return status (from GIM)  */
```

There are two primary purposes for issuing this call. First, the argument "lgth" tells the GIM how much space to allocate for that pseudo-list in a free storage area. The free storage area is part of a data structure which is connected with the logical channel specified in the second argument. The free storage area is pageable and is located in a segment belonging to the GIM.

The second major purpose for issuing this call is to return to the DCM an identification number or name to be used in referring to the list in later calls. The "id" is some unique number concatenated with the logical channel number, and thus the "id" gives the GIM such information as what CDT to use in altering portions of a particular list.

All of the lists generated for a particular logical channel are kept in a structure which is part of the Logical Channel Table (LCT). The LCT, maintained by the GIM, keeps track of the lists as they are created and destroyed, and it makes a record of their size, composition and location in storage.

Both the LCT and the bits of rtn_stat for this call are explained in the summary of the GIM calls and data bases.

Use of the Change-List Call to the GIM

There is no call to the GIM which was specifically designed to take a new and completely empty list and fill it with freshly formed pseudo-words. Instead, the call to modify an already existing list was defined so that the conversion of a null or empty list to a new list would go smoothly and conveniently. Thus an explanation of the "change" calls to the GIM will also show how a newly defined list can be filled in from scratch. The basic change call to the GIM looks like this:

```
change$list (id, indx, hilo, rtn_stat, changesp [,changesp])

dcl  id bit (24),                  /* list identification */
     indx fixed bin (12),          /* index number of first change */
     changesp ptr,                 /* pointer to change structure */
     hilo bit (3),
     rtn_stat bit (36);            /* return status (from GIM) */
```

In the arguments to this call "id" is the list identification number passed back from the GIM when this list was defined via the "define$list" call.

The second argument "indx" requires some further explanation. The term index, as used in talking about the GIM lists and pseudo_lists, always refers to the position of an element in the list relative to the first item in that list. So, the third element in a list would have an index of 3, and the highest index possible in a list is the value of the argument "lgth" when the list was defined.

The "hilo" argument relates the status of edits on active lists. It is fully discussed in BF.20.02, Internal Structure of the GIM. The argument "indx" as used here is effectively a pointer to the first element in the list which will undergo change. The changes to be made to this element are all contained within the structure called "changes" which has this declaration:

```
dcl 1 changes based (changesp),
    2 op_type fixed bin(17),        /* type of edit */
    2 n_changes fixed bin(17),      /* number of sub-edits */
    2 change (n_changes),
      3 field fixed bin(17),        /* field number */
      3 value fixed bin(24),        /* value used in edit */
      3 address ptr;                /* data address */
```

Notice that there are brackets around the second "changes" structure pointer in the change$list call. This is used to indicate that there may be more than one "changes" structure as an argument in the call. The first argument is used to change the pseudo-word pointed to by "indx", and each additional structure argument modifies the next sequential pseudo-word in the list. Through this mechanism an entire list might be changed with a single call.

The following example illustrates use of the "define$list" and change calls described earlier. The example creates a list of ten items, and then fills in the pseudo-words for the first two items on the list. The two items are a command DCW and a literal DCW. The relationships between the fields within the DCW's and their values are contained in the CDT tables for the adapter in question.

```
example: begin;

     dcl  list_1 bit (24), status_1 bit (36), status_2
          bit (36), device_index fixed bin (17), lgth fixed bin
          (12), indx fixed bin (12);

     dcl  1 changes based (p),
          2 op_type fixed bin (17),
          2 nchanges fixed bin (17),
          2 change_array (n);
            3 field fixed bin (17),
            3 value fixed bin (24),
            3 address ptr;
```

```
lgth = 10;
call define$list   (list_1, device_index, lgth, status_1);
if status_1 than go to error_1;
n = 3;
allocate changes set (pw_1);

pw_1 -> changes . op_type = 3;
pw_1 -> changes . nchanges = n;
pw_1 -> changes . change_array (1). field = 1;
pw_1 -> changes . change_array (1). value = 3;
pw_1 -> changes . change_array (2). field = 2;
pw_1 -> changes . change_array (2). value = 3;
pw_1 -> changes . change_array (3). field = 3;
pw_1 -> changes . change_array (3). value = 5;

 n = 2;  allocate changes set (pw_2);

pw_2 -> changes . op_type = 5;
pw_2 -> changes . nchanges = n;
pw_2 -> changes . change_array (1). field = 3;
pw_2 -> changes . change_array (1). value = 2;
pw_2 -> changes . change_array (2). field = 5;
pw_2 -> changes . change_array (2). value = "1000010110"b;

 index = 1;

call change$list (list_1, index, status_2, pw_1, pw_2);

if status_2 then go to error_2;
end example;
```

The important point of this example is to understand how the "changes" structures were developed. Space for the first structure was allocated using the pointer pw_1, and then the elements of this structure were gradually filled in. In the first pseudo-word three fields had to be filled in, and each field was filled in with a preset value from the CDT. In the second pseudo-word (under control of pw_2) one of the fields is set with a preset value, and the other is filled with the literal 1026 (octal).

As the example shows, it would be quite laborious to program the GIM in the manner used above. The main problem is in generating the required change structures. To aid the DCM in creating these structures, an I/O Command Translation module has been designed. This module is described in section BF.20.07 of the manual.

Activation of a Pseudo List

Now that the DCM has the ability to define lists and create the entries within these lists, its next step will be to activate these lists so that the device can start performing the input-output operations specified by the lists. While the list is active there will undoubtedly be interrupts coming from the device to inform the user about the completion of certain events. Finally, when the I/O activity has been completed the DCM may want to release its pseudo-lists so that it can create new or different ones. The sequence of operations above is handled by three separate calls to the GIM which are described below.

The activation of any channel can only be done through the use of the connect$list call. A connect list (i.e., connect pseudo-list) differs from an ordinary pseudo-list in two respects. First, it usually is composed only of connect pseudo words, and consequently only op_type 2 is normally specified in creating the list. The list is created in the same manner as a DCW list, but the connect list itself is never activated or executed. A connect$list call references one and only one item in the connect list, and this item is then sent to the appropriate connect channel. The format for a connect call is as follows:

```
connect$list (con_id, con_index, rtn_stat [,tra_id, tra_index])

dcl  con_id bit (24),              /* id and index of pseudo word */
     con_index fixed bin (12),     /* .. containing channel command
                                      word */
     tra_id bit (24),              /* id and index at which
                                      channel */
     tra_index fixed bin (12),     /* .. is to be started */
     rtn_stat bit (36);            /* return status (from GIM) */
```

If the item being referenced in a connect call is not being used to activate a list channel, then the tra_id and tra_index arguments are meaningless and can be left out. Once a channel has been activated it will remain so until a terminate status word is received from the GIOC.

Status Return Calls

While the channel is active there will probably be a steady stream of interrupts coming from the GIOC to the Device Interface Module. The incoming interrupt status words are stored in a queue, and it is the responsibility of

the DCM to call the GIM and claim its status words. The DIM is awakened for each interrupt that occurs, and if it does not call for its status words before its status queue overflows, they will be lost. Lost status words normally represent a system malfunction. The exact manner in which the status words are stored and processed is described in Internal Structure of the GIM, BF.20.02.

The specification for the status call is

request$status (device_index, current_status, rtn_stat [,cur_stat])

```
dcl device_index fixed bin (17),    /* user device tag */
    current_status bit (1),         /* ON if current status
                                       is desired */
    rtn_stat bit (36);              /* GIM status word */

dcl 1 cur_stat,
    2 filled bit (1),               /* ON if frame has data in
                                       it */
    2 active bit (1),               /* ON if channel is active */
    2 status_waiting bit (1),       /* ON if more status
                                       waiting */
    2 started bit (1),              /* ON if channel has been
                                       started */
    2 int_id bit (24),              /* ID of list */
    2 int_idx fixed bin (12),       /* index of item in list */
    2 tally fixed bin (12),         /* current DCW tally, if
                                       applicable */
    2 time bit (52),                /* time of interrupt */
    2 stat_length fixed bin (17),   /* length of status array */
    2 stat(stat_length) fixed
      bin (24);                     /* breakdown of status */
```

If the user desires the current status of the GIM with respect to his lists and device, the "current_status" switch must be ON ("1"b). If "current_status" is ON, the first user-supplied status structure, "cur_stat" will be filled with current status. Other structures, if any, will be filled with waiting status words, if any. If one or more int_frame arguments are present, then the intent of the call is to retrieve some status words from the status queue buffers for the device in addition to the above-mentioned information.

In the cur_stat structure, the only element which might need explanation is the status-waiting switch. This switch is set whenever the status queue for the channel is non-empty. When this condition is detected, another status call to the GIM should be made to retrieve more status words from the buffer.

Each cur_stat structure is a type of snapshot of the state of the logical channel when the interrupt occurred. The list id returns in int_id, and the index number returns in int_idx.

The important point to note here is that the time of the generation of a status word and the time of servicing an interrupt are not necessarily the same nor necessarily close. It is possible, due to the asynchronous nature of the GIOC, for the index to change substantially between a status store and an interrupt frame snapshot.

The array stat (*) in the interrupt frame is filled in during the analysis of the raw status word. The fields and values within those fields specified by cdt (1) are checked, and if a match is found, an element in stat (*) is filled in with the corresponding status value.

As an example, consider a raw status word which has 8 bits of information, and is equal to "11101011"b. Suppose also that bits 1 ... 8

field 1 = bits 1,2	field 2 = bits 3,8	field 3 = 3,4,5,6,7
f1 value (1) = 00	f2 value (1) = 00	field_action for f3
f1 value (2) = 01	f2 value (2) = 10	is literal substitution
f1 value (3) = 10		
f1 value (4) = 11		

Now, the procedure which processes this raw status word first looks at field 1 and tries to find a match. It does, and sets stat (1) = 4. The procedure then checks field 2 and sets stat (2) = 0 because no match was found. Finally, it takes the literal specified in field 3 and sets stat (3) = "10101"b.

The only status word which has a special meaning to the GIM is the terminate status condition. If the GIM sees a terminate word go by, it makes a note that the channel is no longer active, it releases the space allotted for the DCW lists, and then is ready to activate a new list for that logical channel.

Releasing a List

Once the DCM has determined that a list is no longer necessary it can release that list (or all its lists) by issuing this call

```
define$release (id, terminate_sw, rtn_stat)

dcl id bit (24),                /* ID of list to be released */
    terminate_sw bit (1),       /* ON if releasing all lists */
    rtn_stat bit (36);          /* GIM return status */
```

If the terminate switch is OFF then only the one list specified by id is released. If terminate_sw is ON, then all lists belonging to that logical channel are released. Also, all data areas associated with this device are released and the Logical Channel Table is scrapped. In other words, one terminates use of a device by calling the GIM at the define$release entry with "terminate_sw" ON. Any id of any defined list will suffice for total termination calls.

Global Changes and Copies of Lists

The GIM calls outlined in the above paragraphs are the only calls absolutely necessary in the running of the GIOC. The two following calls are provided for the convenience of the DCM.

```
change$global (id, indx, lgth, mstructp, hilo, rtn_stat,
               changesp [, changesp])

dcl id bit (24),                /* id of list to be changed */
    indx fixed bin (12),        /* starting index within id */
    lgth fixed bin (12),        /* length of block to be changed */
    mstructp ptr,               /* pointer to mask generator */
    changesp ptr,               /* pointer to change structures */
    hilo bit (3)
    rtn_stat bit (36);          /* return status (from GIM) */
```

The argument "mstructp" points to a structure with the same declaration as the "changes" structure, and is used to generate a pair of pseudo-words which will be used in searching through the list specified by "id". One of these words is generated in the same manner as any other pseudo-word. The other pseudo-word is generated by "or"ing together all of the field_masks specified in "mstructp". With this pair of words a mask-value search is started at list "id" location "indx", and when a match is made with some element in the list, then that element is changed using the first changes structure, and each succeeding element is changed according to the optional arguments changesp_1, changesp_2, etc.

With this type of search available the user can now use six bits of the 84 bit pseudo-word as tags to flag certain DCW's for special attention.

When all changes are made, the search is resumed using "mstruct". If another match occurs, the changes are repeated. This process continues in the list until the item with the index of "indx + lgth-1" is searched. In no case are any changes made beyond item (indx + lgth-1).

The call "change$copy", specified below, takes a contiguous block of pseudo words from locations "fidx" to "fidx + fsize-1" in list "fid" and places them in list "tid" at location "tidx",tidx+1,etc. If "tsize" is greater than "fsize", a pseudo element is added to effect a transfer around the extra words. If "fsize" is greater than "tsize", the call is in error.

```
change$copy (fid, fidx, fsize, hilo, rtn_stat, tid, tidx,
             tsize, [, tid, tidx, tsize])

dcl fid bit  (24),              /* id of list from which to copy */
    fidx fixed bin (12),        /* index of block to be copied */
    fsize fixed bin (12),       /* size of block to be copied */
    hilo bit (3),
    rtn_stat bit (36),          /* return status (from GIM) */
    tid bit (24),               /* id of list to be copied into */
    tidx fixed bin (12),        /* index of block to be modified */
    tsize fixed bin (12);       /* size of block to be modified */
```

Summary

This paper has described all the calls necessary for running the GIOC through the facilities of the GIM. A summary of the calls, data bases and status returns is contained in BF.20.03 and BF.20.05. These calls can be used alone, or they can be used with the aid of the I/O Command Translator (BF.20.07). Knowledge of which calls to make and at what time requires a thorough knowledge of how to run a device and what the contents of that device's class driving table are. The calls specified are flexible and allow the DCM writer to use all the features of the GIOC without compromising the security of the system.