TO: MSPM Distribution
FROM: P. G. Neumann
DATE: January 10, 1968
SUBJ: BF.2.23, 2.25, 2.26

The attached copies of BF.2.23, 2.25 and 2.26 represent minor modifications of the existing published documents.

Published: 01/10/68
(Supersedes: BF.2.23, 08/14/67)

Identification

The Attachment Module
R. C. Daley and S. I. Feldman

Purpose

This section describes the Attachment Module. The Attachment Module is called by Device Strategy Modules (DSMs) to handle attach, divert, revert, detach, and invert calls. There are also entries to handle the trap quits and trap hangup order calls and an entry to find out the name of the Registry File with highest level for the device. This section also describes the I/O Registry Files, which are the principal data base of the Attachment Module.

Introduction

The Attachment Module is called to do standard processing of certain outer calls for DSMs. This module is basically responsible for setting up the communication with the Device Control Module (DCM) in the Device Manager Process (DMP), for splicing modules in above the DSM, and for pushing down and popping up paths in response to divert and revert calls, respectively. (In this section, it will be assumed for sake of convenience that the outer module that the DSM wishes to call is a DCM; the DSM could actually call any outer module. The Attachment Module also handles the detachment of the DSM and DCM and, in response to invert calls, deletes paths that have been pushed down but will never be popped up.

This section describes the I/O Registry Files, gives a brief discussion of the Inter-process Communication Block, and then describes detailed handling of the eight entry points to the Attachment Module:

```
attm$attach
attm$divert
attm$revert
attm$detach
attm$invert
attm$trap_quits
attm$trap_hangup
attm$get_rf
```

The I/O Registry Files

The I/O Registry Files describe all of the devices that may be connected to a given system. These files are linked together to indicate the connections between devices. Each Registry File (RF) contains a level number. A level 1 RF corresponds to a GIOC channel; a level 2 RF corresponds to a device connected direcontroly to a GIOC channel, etc. An example of a Registry File chain is the set of files describing a printer attached to a remote computer. The level 3 file describes the printer, the level 2 file describes the remote computer, and the level 1 file describes the GIOC channel and hardware-connected data set.

The Registry Files are organized into directories. The directories are all accessed via the Registry File Directory Directory. It is expected that the directories and the files within them will have many names to allow different ways of specifying a device. In the following, the directory name will be called the "type" and the file name within the directory will be called the "name". The "down" direction is toward the GIOC channel (lower level numbers) and the "up" direction is toward higher level numbers.

The following is a declaration of a Registry File:

```
dcl 1 rf based(rfp),
  2 level fixed bin(35),        /*level=1 for a GIOC channel, 2 for
          "                       a device connected directly to
          "                       a GIOC channel, etc.*/
  2 force_udmp bit(1),          /*if 1, force the use of a universal
          "                       device manager process*/
  2 in_use_switch bit(36),      /*set ON at attach time and OFF
          "                       at detach time*/
  2 hangupable bit(1),          /*if ON, device can hang up*/
  2 logchans bit(1),            /*if ON, the down_names for this
          "                       device are to be filled in by a call
          "                       to the hardcore ring to get the present
          "                       RF name corresponding to the logical
          "                       channel names. If this bit is ON,
          "                       no more RFs are to be searched.*/
  2 allocate bit(1),            /*if ON, Reserver should be called
          "                       with each resource_name as argument.*/
  2 temp_link bit(1),           /*connection with next file is
          "                       temporary. Blank out down_name
          "                       entries upon detachment*/
  2 nup fixed bin(35),          /*number of entries in up array*/
  2 ndown fixed bin(35),        /*number of entries in down array*/
  2 ndev fixed bin(35),         /*number of entries in devices array*/
  2 ntypes fixed bin(35),       /*number of entries in att_types array*
```

```
  2 present_type_index fixed bin(35), /*index in att_types array of
          "                            type with which device was last
          "                            attached*/
  2 down_slot fixed bin(35),  /*position of upname for this file
          "                            in up array of next registry file*/
  2 alloc_type char(32),      /*use this type in calls to the
          "                            Reserver alloc$resource
          "                            entry*/
  2 lock bit(144),            /*for locking RF when threading
          "                            or deleting behavior log entries or
          "                            modifying the profile.*/
  2 up(rfp->rf.nup),          /*registry files pointing to this one*/
    3 uptype char(32),
    3 upname char(32),
  2 devices(rfp->rf.ndev),    /*entries for devices associated
          "                            with this registry file*/
    3 resource_name char(32), /*name used in calls to the Reserver
          "                            and the Device Assignment Module*/
    3 profile_relp bit(18),   /*relp to device profile for this
          "                            device*/
    3 profile_length fixed bin,  /*number of bits in this profile*/
    3 oldest_log_relp bit(18), /*relp to oldest entry in behavior log*/
    3 newest_log_relp bit(18), /*relp to most recent entry in
          "                            behavior log*/
    3 nlog fixed bin,         /*number of entries in behavior log*/
    3 device_type fixed bin(35),
  2 att_types(rfp->rf.ntypes),/*special information for each type
          "                            by which this device may be known*/
    3 type_name char(32),
    3 ccm_type char(32),      /*type of CCM to be spliced in above
          "                            the DSM*/
    3 trace_down bit(1),      /*if ON, trace down to next registry
          "                            file.  Otherwise, stop here*/
    3 alloc_down bit(1),      /*if ON, must call Reserver to
          "                            allocate a device of type
          "                            down_type, and use returned
          "                            resource_name as down_name(1).
          "                            In either case, find next RF by
          "                            using down_type and down_name(1)*/
    3 look_only bit(1),       /*keep tracing down to other RFs
          "                            under trace_down control, but
          "                            only to compute CCM typename*/
    3 down_type char(32),     /*used as described above*/
    3 down_name(rfp->rf.ndown) char(32), /*used as described above*/
    3 logical_channel (rfp->rf.ndown) char(32),  /*array of
          "                            names to be used in call to get present
          "                            equivalent RF name from
          "                            info in DCT.  Used only if the
          "                            logchans bit is on*/
    3 extra_mode char(32),    /*character string to be
          "                            concatenated with mode to be
          "                            passed to DCM*/
    3 dcm_type char(32),      /*used as type in attach call to
```

```
          "                          DCM if trace_down is OFF or
          "                          look_only is ON*/
    3 dcm_name char(32),           /*used as ioname2 of attach call to
          "                          DCM if trace_down is
          "                          OFF or look_only is ON*/
  2 free_storage area((15000));
/*

*/
dcl 1 rf_ro based(p),              /*special Registry File. There is a
          "                          file of this format associated with
          "                          each regular RF, with name equal to
          "                          the name of the normal RF concatenated
          "                          with "_ro". This file contains
          "                          certain data that must be protected
          "                          against tampering and is therefore
          "                          read-only to most users.*/
  2 pdt_name char(32),             /*name of PDT in DMP*/
  2 udmp_user_id char(50);         /*user_id of universal device manager
          "                          for this device, if any*/
```

As is clear from the above, a Registry File contains a large number of switches and character strings. The discussion of the various entry points to the Attachment Module explains the use of the various parts of the Registry File.

The "up" array of a RF contains the list of names of Registry Files with down_type and down_name equal to the name of the given file. The kth uptype and upname equals the name of the file with down_type and down_name equal to the name of this file and with down_slot equal to k.

A Registry File may represent several separate devices which, for various reasons, it is convenient to consider as a single device. An example of such a grouping is a full-duplex typewriter channel, which requires two GIOC channels to control one typewriter; such a pair of channels would have a single level 1 RF. The "devices" array in the Registry File is designed to handle such cases.

Since directories may have several names, a given RF may be reached with several different types. Different chaining of files and different calls to the DCM may be desired for different types. Information on the name of the driving table for use by the Code Conversion Module (CCM), if any, and the method of finding the next lower RF, if any, is kept in the att_types array.

There are two different ways the next RF may be found. If an alloc_down switch in the att_types array is OFF, then the corresponding down_type and down_name entries specify the next

Registry File. This type of linking is used when the association between devices is known. If an alloc_down switch is ON, then a device of type equal to the corresponding down_type must be allocated. This form of allocation is used, for example, to allocate a 7-track tape drive on which to mount a particular tape; the user does not care which particular drive is used.

Certain entries in the RFs are of interest to outer modules. Specifically, the resource_names are needed by DCMs to make assign calls to the GIM. The device_types and profile pointers are needed by outer modules which need descriptions of devices. An I/O Registry File Maintainer will be supplied to fill such needs.

The Inter-process Communication Block

The per-ioname segment (IS) of each DSM contains an Inter-process Communication Block (ICB) which contains information used by the Attachment Module, Request Queuer, and Driver. It contains event channel names, switches to indicate functions to be performed by the Dispatcher, ionemaes, Registry File names, two lock lists, and various other pieces of information. The following is a declaration of the ICB:

```
dcl 1 icb based (p),                  /*inter-process communication block*/
  2 queue_lock_list bit(144),         /*standard lock for request queuing*/
  2 iocall_event bit(70),             /*event channel name*/
  2 dmp_proc_id bit (36),             /*device manager process id*/
  2 dmp_user_id char(50),             /*user id of dmp if not private*/
  2 private_dmp bit(1),               /*1 if a private DMP was created*/
  2 quit_event bit(70),               /*event name*/
  2 restart_event bit(70),            /*name of event channel to be signaled
          "                             to restart path in DMP without
          "                             passing an outer call*/
  2 reset bit(1),                     /*set to 1 to cause a reset
          "                             of all calls in request queue
          "                             when next restart is done*/
  2 invert bit(1),                    /*set to 1 to cause diverted paths
          "                             in DMP to be detached*/
  2 invert_event bit(70),             /*name of event channel to be
          "                             signaled when inversion complete*/
  2 divert bit(1),                    /*set to 1 to cause present iopath
          "                             to be quit*/
  2 divert_event bit(70),             /*name of event channel to be
          "                             signaled when diversion complete*/
  2 trap_quits bit (1),               /*if 1,signal if quit occurs
          "                             on device*/
  2 overseer_trap_hangup bit(1),      /*if 1, signal overseer if
          "                             hangup occurs on device*/
  2 trap_hangup bit(1),               /*if 1, signal if hangup occurs on
```

```
        "                               device*/
  2 quit_id bit (36),                  /*id of process to be signaled on quit*/
  2 overseer_id bit(36),               /*process id of overseer*/
  2 hangup_id bit(36),                 /*id for process to be signaled
        "                               when device hangs up*/
  2 quit_report_event bit(70),         /*event signaled if device quit*/
  2 overseer_hangup_report_event bit(70),  /*event to be
        "                               signaled if hangup occurs*/
  2 hangup_report_event bit(70),       /*event to be signaled if
        "                               device hangs up*/
  2 diverted bit(1),                   /*1 if this ioname has been
        "                               diverted*/
  2 divert_type bit(1),                /*when diverting, set to 1 if
        "                               the two ioname arguments are
        "                               equal*/
  2 alloc_down bit(1),                 /*how this registry file was reached.
        "                               If ON, device of given type was
        "                               allocated and name returned.
        "                               Otherwise, name came from description
        "                               argument of call.*/
  2 dsm_rf_type char(32),              /*type of first RF (highest level)*/
  2 dsm_rf_name char(32),              /*name of first RF*/
  2 dcm_type char(32),                 /*type to be used in attach
        "                               calls to the DCM*/
  2 dcm_description char(32),          /*description to be used in attach
        "                               calls to the DCM*/
  2 nchar_dcm_mode fixed bin(17),      /*number of characters
        "                               in dcm_mode*/
  2 dcm_mode_relp bit(18),             /*relp to character string
        "                               equal to mode of DCM*/
  2 old_dsm_ioname char (32),          /*previous dsm ioname*/
  2 new_is_name char(32),              /*for use when diverting.
        "                               Name of new
        "                               per-ioname segment*/
  2 dcm_ioname char(32),               /*for possible future use in
        "                               handling NODMP mode*/
  2 old_dcm_ioname char(32),           /*same as above*/
  2 icb_lock_list bit(144),            /*standard lock*/
  2 invert_proc_id bit(36),            /*response event for invert*/
  2 divert_proc_id bit(36);            /*response event for divert*/
```

The Process Dispatching Table

The Process Dispatching Table (PDT) describes the devices that may be controlled by a given Device Manager Process, and is therefore of interest mainly to the Dispatcher. However, if a private DMP is to be created, the Attachment Module must create the DMP. Also, the Attachment Module must know the declaration of the PDT in order to find the DMP's process id and the name of the reassign event channel. Therefore, for convenience, a declaration of the PDT is included here. For a more detailed discussion of the PDT, see BF.2.25

```
dcl 1 pdt based(p),                 /*Process Dispatching Table*/
  2 init_proc char(32),             /*name of procedure to be
        "                             called for initialization.
        "                             Equal to "disp$init"*/
  2 dmp_proc_id bit(36),            /*id of this  Device Manager
        "                             Process*/
  2 reassign_event bit(70),         /*event channel to be signaled
        "                             when device is assigned or
        "                             unassigned to this process*/
  2 creator_id bit(36),             /*id of process that created this
        "                             Device Manager*/
  2 init_done_event bit(70),        /*event channel to be signaled when
        "                             initialization of this process is
        "                             complete.*/
  2 current ptr,                    /*pointer to element of routes
        "                             for device for which work
        "                             is being done at present*/
  2 pdt_name char(32),              /*name used by other processes to
        "                             find PDT*/
  2 dtabp ptr,                      /*pointer to Driver"s driving
        "                             table*/
  2 disp_ptr,                       /*pointers to entry points of
        "                             the Dispatcher*/
    3 reassign ptr,
    3 iocall ptr,
    3 reenable ptr,
    3 restart ptr,
    3 quit ptr,
    3 hardware ptr,
  2 nroutes fixed bin(17),          /*number of entries in routes array*/
  2 routes(n),                      /*an entry for each device which
        "                             may be assigned to this process.
        "                             n = pdt.nroutes*/
    3 type char(32),                /*type of resource*/
    3 resource_name char(32),       /*resource_name for this device*/
    3 user_id char(50),             /*user to whom device is assigned*/
    3 ioname char(15),              /*DCM ioname, a unique character string*/
    3 pibp ptr,                     /*pointer to PIB for this DSM*/
    3 icbp ptr,                     /*pointer to ICB for DSM*/
    3 tbsp ptr,                     /*pointer to Transaction Block
        "                             segment in user"s group
        "                             directory*/
    3 att_stack ptr,                /*pointer to entry in attach_stack
        "                             area for pushed-down DCM*/
    3 iocall_event bit(70),         /*event to be signaled by DSM
        "                             for iocalling, resetting,
        "                             inverting, and diverting*/
    3 restart_event bit(70),        /*signaled to restart a path
        "                             in external quit condition*/
    3 hardware_event bit(70),       /*event channel signaled when
        "                             interrupt received from device*/
```

```
    3 quit_event bit(70),      /*event to be signaled to stop
        "                        device and prepare for a divert*/
    3 reenable_event bit(70), /*signaled when auxiliary
        "                        chain or TBS is unlocked*/
    3 device_absent bit(1),    /*1 if device not present*/
    3 assigned bit(1),         /*1 if device assigned to this
        "                        process*/
    3 attached bit(1),         /*1 if attach call has been
        "                        issued*/
    3 ext_quit bit(1),         /*1 if device in external quit
        "                        condition*/
    3 int_quit bit(1),         /*1 if device in internal (hardware)
        "                        quit condition*/
  2 attach_stack area((10000));/*area into which blocks are
        "                        allocated for diverted paths*/
/*


*/
dcl 1 att_thread based(p),     /*declaration of block to be
        "                        allocated into att_stack
        "                        area for pushing down of
        "                        DCMs*/
  2 ioname char(15),           /*DCM ioname*/
  2 iocall_event bit(70),      /*event channel name*/
  2 reenable_event bit(70),    /*event channel name*/
  2 pibp ptr,
  2 icbp ptr,
  2 status,
    3 attached bit(1),
    3 ext_quit bit(1),
  2 next ptr;                  /*points to next block in thread
        "                        of pushed-down DCMs*/
```

When necessary, the Attachment Module calls the Mode Handler (see BF.2.27) to interpret mode and disposal arguments of calls. Therefore, the DSM does not need to make these calls. The bmode string in the PIB is updated whenever a mode argument is interpreted.

Attach Call Processing

The Attachment Module has an entry point to perform most of the processing needed for a DSM to handle an attach call. The call is:

```
call attm$attach(pibptr,mode,dcmstatus,sfmstatus,cstatus);
dcl mode char(*),          /*fourth argument of attach
                             call*/
    dcmstatus bit(144),    /*status from localattach call
     "                       sent to DCM*/
    sfmstatus bit(144),    /*status from attach call to SFM,
     "                       if SECTIONAL mode specified*/
    cstatus bit(18);       /*status returned by Attachment
                             Module*/
```

In response to this call, the Attachment Module traces down the Registry Files corresponding to the device or devices implied by the type and description arguments of the attach call received by the DSM. As necessary, devices are allocated, depending on switches in the various RFs. A private DMP is created only if the PRIVATE mode is specified in mode and if the use of a Universal Device Manager Process is not forced by the Registry Files. The attach call is passed to the DCM with type and description arguments found in the last RF and a mode argument computed using strings found in the RFs and passmode (the string returned by the Mode Handler in step 1 below).

The description argument is considered to be a sequence of components delimited by slash ("/") characters. These components are used as Registry File names when a device is to be allocated (see below). A subroutine of the Attachment Module will be used to break the string into components and to delete blanks.

It is convenient to define a few temporary variables in the following discussion. Let A be a switch which indicates how a Registry File was reached. If ON, a call was made to the Reserver to allocate a device of a particular type and the file name was returned; if OFF, the name was known without performing such a call. Let NAME be the name of the RF and TYPE its type. Let UPTYPE and UPNAME be the last values of TYPE and NAME, and let N be an integer indicating the position in the "up" array into which UPTYPE and UPNAME are to be stored. Let CCMTYPE be the name of the driving table to be used by the CCM. Let DCMMODE be the mode to be used in attaching the DCM. Let HANGUPABLE be a switch indicating whether the device can hang up. Let LASTTYPE be the type of the last Registry File through which the Attachment Module traced down (see below for explanation of terms). Let RESOURCE be the first resource_name in that file and FORCEUDMP be the force_udmp bit in that file.

In the following, "description" is a character string of length 32 equal to the corresponding argument in the attach call. This string is taken from pib.ioname2.

In response to the attm$attach call, the following steps are taken:

1. Set cstatus equal to zero and call the Mode Handler (see BF.2.27) to interpret mode. Two arguments are returned: passmode (a character string of modes to be passed on to the DCM) and bmode, a 72-bit bit string which is a summary of the modes at this time. Both of these return arguments will be used below. If there is any error in the mode interpretation, set bit 10 of cstatus and return. Otherwise, store bmode in pib.bmode.

2. Allocate the ICB in area pib.ioarea and initialize it. All items in the ICB should be zero except the values of "n" in the two lock lists should be set to the appropriate values.

3. Initialize: Blank out UPTYPE, UPNAME, CCMTYPE, and DCMMODE. Set TYPE and icb.dsm_rf_type equal to pib.typename. Set HANGUPABLE, dcmstatus, and sfmstatus equal to zero.

4. This and the next step find the next (first) Registry File. If the next (first) component of description is not null, go to step 5. Otherwise, a device of type TYPE must be allocated. This is done by the following call to the Reserver:

```
call alloc$type(TYPE,NAME,status);
```

The Reserver will, if a device is available, return the name of the device in NAME. If no device is available, set bit 3 of cstatus and go to step 27. If an allocation is made, set A ON to indicate how the allocation was made. If there is no present Registry File (i.e., this is the first time step 4 has been reached), store NAME in icb.dsm_rf_name, Set the alloc_down switch in the ICB ON. If there is a present RF, store NAME in rf.att_types.down_name(rf.present_type_index). Go to step 6.

5. If the component of description checked in step 4 is not null, set NAME equal to that component and set A OFF to indicate how that name was found. If there is not a current Registry File, set icb.alloc_down OFF and store NAME in icb.dsm_rf_name. If there is a current registry file, store NAME in rf.att_types.down_name(rf.present_type_index).

6. Find and lock the proper Registry File:

   a. If NAME is blank, set bit 2 of cstatus and go to step 27: the Registry Files have not yet been linked. This can happen if an attachment is attempted before a dialup has occurred on a typewriter.

   b. Search the Registry File Directory Directory for a directory with name TYPE. If such a directory is not found or is not accessible to this user, set bit 1 of cstatus and go to step 27.

c. Search that directory for a file name NAME. If no such file exists or if it is not accessible to this user, set bit 1 of cstatus and go to step 27.

7. If the logchan bit in the RF is ON, go to step 8. If A is OFF and the allocate switch in the RF is ON, then the Reserver should be called to allocate the devices associated with the file by making the following call:

```
call alloc$resource(alloc_type,rf.devices.resource_name(1),status);
```

If the allocation fails, set bit 3 of cstatus and go to step 27.

8. Search the att_types array for a "type" entry equal to TYPE. If none is found, set bit 5 of cstatus and go to step 27. If an entry with the proper type is found, do the following:

a. Store the index of that type in present_type_index.

b. Store the present process_id in in_use_sw.

c. If this is the first RF found, store blanks in all the uptype and upname entries. If this is not the first RF found, store UPTYPE and UPNAME in rf.up(N).uptype and rf.up(N).upname, respectively.

d. Set DCMMODE equal to DCMMODE || "/" || rf.extra_mode.

e. Set CCMTYPE equal to CCMTYPE || rf.ccm_type. Remove embedded blanks from CCMTYPE. (Assume that rf.ccm_type and CCMTYPE are left adjusted and padded on the right with blanks).

f. Set HANGUPABLE equal to HANGUPABLE || rf.hangupable.

g. If rf.ndown is equal to one and rf.logchans is OFF, go to step 9.

h. If rf.ndown is less than one, set bit 12 of cstatus and go to step 27.

i. If rf.logchans is OFF, go to k.

j. Call the DCTM with each of the ndown logical channel names and store the returned value the corresponding element of the down_name array. If any of these calls fails, set bit 3 of cstatus and go to step 27.

k. Find the ndown RFs pointed to by the down_type and down_names in the present RF. If the allocate bit in one of those files is ON, call alloc$resource using the first resource_name in that file as name argument and down_type (above) as type. If any of these RFs do not exist or if any

of the allocations fail, deallocate any devices allocated so far in this part of the step, set bit 3 of cstatus and go to step 27.

1. Otherwise, go to step 16.

9. If the trace_down switch is OFF, store dcm_type in icb.dcm_type and store dcm_name in icb.dcm_description. Set LASTTYPE equal to TYPE and set RESOURCE equal to the first resource_name in the present RF. Set FORCEUDMP equal to the force_udmp bit in the file and go to step 16.

10. If the look_only switch is ON, go to step 12. Otherwise, set N equal to down_slot, UPTYPE equal to TYPE, UPNAME equal to NAME, and TYPE equal to down_type. These assignments are necessary to prepare to examine the next RF.

11. If the alloc_down switch in the RF is ON, go to step 4 to allocate the device. Otherwise, set NAME equal to down_name(1), set A OFF, and go to step 6.

12. If the look_only switch is ON, continue examining Registry Files, but only to compute DCMMODE, CCMTYPE, and HANGUPABLE. It is assumed that the lower RFs are already linked. Set icb.dcm_type equal to dcm_type and icb.dcm_description equal to dcm_name. Set LASTTRACE equal to TYPE and set RESOURCE equal to the first resource_name in the present RF. Set FORCEUDMP equal to the force_udmp bit in the file.

13. If rf.ndown is not equal to 1, set bit 2 of cstatus and go to step 27. Otherwise, find the RF with type equal to rf.att_types(rf.present_type_index).down_type and name equal to rf.att_types(rf.present_type_index).down_name(1). If either of these character strings is blank, set bit 2 of cstatus and go to step 27. If no such file exists and is accessible to this user, set bit 1 of cstatus and go to step 27. If the RF found has a zero in_use_sw, then set bit 2 of cstatus and go to step 27.

14. Calculate DCMMODE, CCMTYPE, and HANGUPABLE:

a. Set DCMMODE equal to DCMMODE || "/" || rf.att_types(rf.present_type_index).extra_mode.

b. Set CCMTYPE equal to CCMTYPE || rf.att_types(rf.present_type_index).ccm_type. Remove embedded blanks.

c. Set HANGUPABLE equal to HANGUPABLE | rf.hangupable

15. If rf.att_types(rf.present_type_index).trace_down is ON, go to step 13.

16. Store the number of characters in DCMMODE in icb.nchar_dcm_mode, and then allocate a string of that length in

pib.ioarea. Store DCMMODE in that string and store a relative pointer to that string in icb.dcm_mode_relp.

17. If the PRIVATE mode is specified in bmode and if FORCEUDMP is OFF, go to step 18. Otherwise, a Universal Device Manager Process is supposed to be used. The name of the group containing that DMP and the name of the PDT for that process (in that group's group directory) are found in a special file. That file is found in the directory with name equal to LASTTYPE. The name of the file in that directory equals RESOURCE || "_ro" (for "read-only", the attribute of the file). Store the user id of the DMP in icb.dmp_user_id and set icb.private OFF. Call the Device Assignment Module to assign the Universal Device Manager Process as the control user of the device:

```
call ioam$set_control(LASTTYPE,RESOURCE,udmp_user_id,error);
```

Get a pointer to the PDT for the DMP, which can be found using the naming algorithm described above. Keep this pointer for use in step 20. Go to step 19.

18. If a private DMP is desired, the following steps are taken:

a. Create a Process Dispatching Table segment for a new DMP as a branch of the present group directory. Set nroutes equal to 1, set routes(1).type equal to LASTTYPE, and set routes(1).resource_name equal to RESOURCE. Set pdt.init_proc equal to "disp$init".

b. Create an event channel and store its name in pdt.init_done_event and store the present process id in pdt.creator_id.

c. Store the unique character string created in step a above (the entry name of the PDT) in pdt.pdt_name.

d. Create the DMP by a call to create_proc (see BJ.2).

e. Wait for the response event to be signaled, and then destroy that event channel.

f. Set icb.private_dmp ON.

19. Create a link in the user's group directory to the per-ioname segment (IS) with name RESOURCE. This link is used by the Dispatcher in the DMP to access the IS.

20. Signal the reassign event for the DMP. The event name is found in the Process Dispatching Table (PDT) in the DMP. This signal will cause the Dispatcher to prepare for an localattach call for the device from this user.

21. Create an event channel. If icb.private is OFF, give user icb.dmp_user_id access to the channel. Call the Request Queuer (see BF.2.24) to pass a localattach call to the DCM. Use icb.dcm_type as the type argument, icb.dcm_description as the description, and DCMMODE || passmode as the mode argument. Use the event channel just created as the completion event. Upon return from the Request Queuer, wait for the completion event to be signaled. Destroy the event channel. Use the Transaction block index returned by the Queuer to make a call to rq$get_chain for the localattach call. Store that status in the status argument of the call to the Attachment Module.

22. Delete the link to the IS. If the attachment failed, set bit 4 of cstatus and go to step 27.

23. In order for the quit and restart mechanism to work, the Overseer must have available a list of devices and certain associated event channel names available. The io_control procedure (see BF.3.01) is the interface between the Overseer quit and restart mechanism and the I/O System. The following call is made by the Attachment Module to inform io_control of the new device and to get certain information from the ICB and to put other information into the ICB:

```
call io_control$attach(pib.ioname1,type,description,icb.overseer_id,
     icb.dmp_proc_id,icb.quit_event,icb.restart_event,
     HANGUPABLE,icb.overseer_hangup_report_event,cstatus);
```

If HANGUPABLE is OFF or if the hangup report event is zero, set icb.overseer_trap_hangup bit OFF; otherwise, set it ON.

24. If CCMTYPE is blank, then go to step 25. Otherwise, the DSM uses a Code Conversion Module (CCM), and the second driving table pointer (pib.dtabp2) must point to the appropriate driving table. Therefore, the following call is made:

```
call atm$change_dtab(pib.ioname1,2,CCMTYPE,0,"0"b,cstatus);
```

25. If bmode specifies the SECTIONAL mode, the Sectional Formatting Module (SFM) (see BF.8) must be spliced in immediately above the DSM. This is done by renaming the DSM's switchpoint and then attaching the SFM. First, a unique name is created (by a call to unique_chars(unique_bits), see BY.15.01). Then the following call is made:

```
call atm$setioname1(pib.ioname1,unique_name,cstatus);
dcl unique_name char(15),
    cstatus bit(18);
```

The SFM is then attach with the same mode that the DSM received:

```
call attach(pib.ioname1,"sfm",unique_name,mode,status);
dcl status bit(144);
```

Finally, set the ioname1 entry of the DSM's PIB equal to the unique_name.

26. Return to the DSM.

27. In case of error, call the internal cleanup procedure to restore the RFs to their previous condition. To do this, make the following call:

```
call cleanup(icbptr,bdisp);
dcl icbptr ptr,
    bdisp bit(72);
```

For bdisp, use a bit string that would represent the HOLD/UNLOAD disposal modes. Upon return from cleanup, return to the DSM.

Divert Call Processing

The divert outer call is used to push down an I/O path and to create a fresh one for temporary use. The Attachment Module has an entry point to to handle divert calls. The call is:

```
call attm$divert(pibptr,newioname,mode,cstatus);

dcl newioname char(*),  /*new name of diverted path*/
    mode char(*),       /*mode argument of divert call*/
    cstatus bit(18); /*status for this call*/
```

The third argument is part of the string of modes to be used in establishing the new DSM and DCM. The DSM ioname is changed if newioname is equal to the present ioname of the DSM. The Dispatcher is told to push down the old DCM and to create a new one.

The divert call is passed by the I/O Switch regardless of the lock on the per-ioname segment. No transaction block is allocated by the switch for this call because of interlocking problems.

The following steps are taken in response to the attm$divert call:

1. Zero cstatus and call the Locker to lock the icb (using icb.icb_lock_list).

2. Call the Mode Handler to interpret mode. The Mode Handler returns bmode and passmode. If there was an error in the interpretation of mode, set bit 10 of cstatus, unlock the ICB, and return.

3. If the diverted bit in the ICB is ON, then this path has already been diverted and cannot be diverted again. Set bit 7 of cstatus, unlock the ICB, and return.

4. If pib.ioname1 (the DSM's ioname) equals newioname, then create a unique name and then make the following call:

```
call atm$rename_attach_return(pib.ioname1,unique_name,
     pib.typename,pib.ioname2,status);
dcl status bit(144);
```

This call causes the present ioname to be changed to the unique name and the partial attachment of a new ioname node with the previous name of the present node. Thus, in one step, we have created a new path and saved the present one. The new path has not been fully attached, however. It will be activated by a future attach order call made by the Attachment Module. Set the divert_type bit in the ICB to "1"b and go to 6.

5. If pib.ioname1 is not equal to newioname, set the divert_type bit to 0. Make the following calls:

```
call io_control$lock;

call atm$attach_return(pib.ioname1,newioname,
     pib.typename,pib.ioname2,status);
dcl status bit(144);

call io_control$rename(newioname,pib.ioname1,cstatus);
```

The first call keep io_control's data base, the OIL, locked until the next call to io_control. The next call creates a new ioname but does not pass an attach call to it. By doing this, the Attachment Module has a chance to fix up the new ICB before initializing the new ioname by an attach order call. The third call informs io_control of the new name and causes the OIL to be unlocked.

6. Get a point to the new per-ioname segment by means of a call to atm$get_iosegname. Store newioname in the ioname1 entry of the new PIB, and then copy the typename and ioname2 entries of the present PIB into the corresponding entries of the new PIB.

7. Allocate the ICB in the new IS and lock it. Initialize this ICB by copying the following from the old ICB:

All of the event channel names other than invert_event and divert_event.
All process ids.
The following one-bit items: trap_quit, trap_hangup, overseer_trap_hangup, private_dmp, and alloc_down.
All character strings other than old_dsm_ioname, new_isname, dcm_name, and old_dcm_name.
Copy nchar_dcm_mode into the new ICB and then allocate in the new IS a character string of that length and copy the string pointed to by the dcm_mode_relp. Store a relp to that string into the new dcm_mode_relp.

8. Store the ioname of the old DSM in the old_dsm_ioname entry of the new ICB.

9. Store the name of the new IS in the new_is_name entry of the old ICB. The Dispatcher will use this name to access the new IS.

10. Store the bmode computed in step 2 in the bmode entry of the PIB of the new DSM.

11. Create an event channel and store its name in the old icb.divert_event. Store the present process id in the old icb.divert_proc_id. If icb.private is OFF, give user icb.dmp_user_id access to the event channel. Set icb.divert ON and signal the iocall event. The Dispatcher will push down that part of the path, create a new DCM ioname and a new iocall event,

and then signal the response event. Wait for the response event.

12. Call the Request Queuer to set up a regular localattach call for the DCM in the Request Queue of the new IS. Use icb.dcm_type and icb.dcm_description as the type and description arguments of that call, and use the concatenation of the dcm_mode in the IS and passmode as the mode argument of that call. Wait for the completion event. Destroy the event channel.

13. Set the diverted bit in the old ICB ON.

14. Make the following call:

```
call order(newioname,"attach",null,null,status);
```

This call will force the new DSM to initialize itself by performing all of the steps involved in normal attach processing other than calling the Attachment Module.

15. Unlock both the new and the old ICB.

16. Return to the DSM.

Revert Call Processing

In order to pop up a diverted iopath, the revert outer call is used. After the DSM has done necessary cleanup it makes the following call to the Attachment Module:

```
call attm$revert(pibptr,disp,cstatus);

dcl disp char(*),          /*disposal argument of revert
                             call*/
    cstatus bit(18);
```

The Attachment Module takes the following steps to pop up the DSM and DCM:

1. Lock the ICB.

2. If the diverted bit in the ICB is ON, set bit 7 of cstatus, unlock the ICB, and return: only the most recent iopath may be reverted.

3. If the old_dsm_ioname entry in the ICB is blank, then there is no diverted path to revert; in that case, set bit 6 of cstatus and go to step 11.

4. Call the Mode Handler to interpret disp. The Mode Handler will return passmode, the disposal string to be used in the call to the DCM, and bdisp, a bit string of length 72 which contains a summary of the disposal modes at this node. If the Mode Handler indicates an error, set bit 10 of cstatus and go to step 11.

5. Set up an ordinary detach call with disposal equal to the concatenation of passmode and "/DEV1" in the Request Queue and signal the iocall event for the device. Wait for the return event. When the Driver in the DMP returns to the Dispatcher after handling a detach call, the Dispatcher pops up the next DCM ioname and ioname segment using information in the PDT.

6. Get a pointer to the PIB of the per-ioname segment of the old DSM by making the following call:

```
call atm$get_iosegname(icb.old_dsm_ioname,isname,pibptr,cstatus);
```

Using this pointer, get a pointer to the popped-up ICB.

7. Make the following call:

```
call atm$delete_ioname(pib.ioname1,"1",status);
dcl status bit(18);
```

Upon return to the I/O Switch from this outer call, the Attach Table entry for this ioname and the IS will be destroyed and all related Transaction Blocks released.

8. If the RESET disposal mode is specified in bdisp, set the reset bit in the popped-up ICB. Signal the restart event. The Dispatcher will call driver$restart with the reset bit as argument. If the reset bit is ON, then all pending transactions will be reset (aborted and not to be restarted).

9. If the divert_type bit in the popped up ICB is 1, then make the following calls:

```
call atm$switch_ionames(old_dsm_ioname,
     pibptr->pib.ioname1,cstatus);

call iosw$queue_restart(pib.ioname1,cstatus);
```

These calls exchange the ionames of the nodes and then restart the path that was just popped up. Go to step 11.

10. If the divert_type bit in the popped-up ICB is 0, then make the following calls:

```
call io_control$rename(old_dsm_ioname,
     pibptr->pib.ioname1,cstatus);

call iosw$queue_restart(old_dsm_ioname,cstatus);
```

These calls update the OIL (Overseer Ioname List; see BF.3.01) and restart the popped-up path.

11. Unlock the ICB.

12. Return to the DSM.

Detach Call Processing

When a DSM receives a detach call, it must clean up all pending I/O, including I/O to be done by diverted iopaths, and then detach all of the CCMs, DSMs, and DCMs in these paths. The Attachment Module has an entry to perform these functions. In response to a detach call, a DSM cleans up its pending I/O and then performs the following call:

```
call attm$detach(pibptr,disp,cstatus);
dcl disp char(*),          /*disposal argument
                             of detach call*/
    cstatus bit(18);
```

This call causes the DCM to be detached, and, depending on the disposal modes, devices to be deallocated and reservations to be released.

The following steps are taken in response to the call to attm$detach:

1. Call the Mode Handler to interpret disp. The Mode Handler will return passmode and bdisp, as described above. If there was an error in interpreting disp, set bit 10 of cstatus and return.

2. Call attm$invert to destroy any pushed-down paths.

3. Send a detach call to the DCM in the DMP with disposal argument equal to the concatenation of passmode and "/DEV1".

4. Call the cleanup internal procedure to deallocate devices, cancel reservations, clean up Registry Files, and destroy the DMP if private.

5. Make the following call to remove this ioname from the Overseer ioname List:

```
call io_control$detach(pib.ioname1,cstatus);
```

6. Make the following call to the ATM to delete this ioname upon return to the I/O Switch:

```
call atm$delete_ioname(pib.ioname1,"1"b,status);
```

7. Return to the DSM.

Invert Call Processing

When a DSM receives an invert call, it is supposed to delete all all iopaths that have been diverted for that ioname. The DSM immediately makes the following call to the Attachment Module:

```
call attm$invert(pibptr,cstatus);
dcl pibptr ptr,
    cstatus bit(18);
```

The following steps are taken:

1. Lock the ICB.

2. If the old_dsm_ioname in the ICB is blank, then there are no pushed down paths, so go to step 6.

3. Create an event channel and store its name in icb.invert_event. Store the present process id in icb.invert_proc_id. If icb.private is OFF, give user icb.dmp_user_id access to the event channel. Set icb.invert ON and signal the iocall event. Wait for the response event, and then destroy that event channel. The Dispatcher will call driver$detach for each of the pushed-down DCM ionames.

4. For each ioname in the chain of old_dsm_ionames, perform the following steps:

a. Unlock the DSM's ICB.

b. Unlock the DSM's auxiliary chain.

c. Make the following call for the DSM and CCM ioname:

```
call atm$delete_ioname(ioname,"0"b,status);
dcl ioname char(32),
    status bit(18);
```

This call immediately deletes the given ioname from the AT and destroys the per-ioname segment.

5. Store blanks in the and old_dsm_ioname entry in the current (and only) ICB.

6. Unlock the ICB.

7. Return to the DSM.

Quit Reporting

In order to allow processes to be signaled when a quit is signaled on a device, there is a special "trap_quits" order call. In response to such an order call, a DSM makes the following call to the Attachment Module:

```
call attm$trap_quits(pibptr,proc_id,quit_event,cstatus);

     dcl proc_id bit(36),      /*id of process to
                                 receive event signal*/
         quit_event bit(70),   /*name of channel to be
                                 signaled*/
         pibptr ptr,
         cstatus bit(18);
```

The Attachment Module takes the following steps in response to this call:

1. Check the validity of the call. If the current validation level of the caller is higher than the validation level of the caller of the original attach call for the DSM, set bit 8 of cstatus and return.

2. If either proc_id or quit_event is zero, set icb.trap_quits OFF and return.

3. Store proc_id in icb.quit_id.

4. Store quit_event in icb.quit_report_event.

5. Set icb.trap_quits ON.

6. Return.

Hangup Reporting

In order to allow processes to be signaled when a hangup occurs on a device, there is a special "trap_hangup" order call. In response to such an order call, a DSM makes the following call to the Attachment Module:

```
call attm$trap_hangup(pibptr,proc_id,hangup_event,cstatus);

     dcl proc_id bit(36),
         hangup_event bit(70),
         pibptr ptr,
         cstatus bit(18);
```

In response to such a call, the Attachment Module takes the following steps:

1. Check the validity of the call. If the validation level of the caller is higher than the validation level of the caller of the original attach call, reject the call by setting bit 8 of cstatus and return.

2. If either proc_id or hangup_event is zero, set icb.trap_hangup OFF and return.

3. Store proc_id in icb.hangup_id.

4. Store hangup_event in icb.hangup_report_event.

5. Set icb.trap_hangup ON.

6. Return.

Call to Get Registry File Name

The following call is used by the DSM to find the name of the Registry File with the highest level number for this ioname:

```
     call attm$get_rf(pibptr,type,name,cstatus);

     dcl type char(32),        /*type of the RF*/
         name char(32),        /*name of the RF*/
         cstatus bit(18);
```

The following steps are taken in response to this call:

1. If pib.typename is equal to icb.dsm_rf_type, then set type = icb.dsm_rf_type and name = icb.dsm_rf_name and return.

2. Otherwise, look through the up array of the RF with type icb.dsm_rf_type and name icb.dsm_rf_name for an uptype equal to pib.typename. If one is found, store the uptype in type and in icb.dsm_rf_type, store the upname in name and in icb.dsm_rf_name, and return. If no such uptype is found, set bit 9 of cstatus and return.

Cleanup Internal Procedure

The cleanup procedure is called to restore the RFs to their state before attachment was started, to release reservations of devices, unload devices, and destroy private DMPs. The following call is made to this routine:

```
call cleanup(icbptr,bdisp);
dcl icbptr ptr,
    bdisp bit(72);
```

It is convenient to define a few temporary variable: LOOKONLY, TRACEDOWN, TEMPLINK, DOWNTYPE, DOWNNAME (array), NDOWN, and DOWNSLOT, which are copied out of a Registry File before an attempt is made to delete it.

The following steps are taken in response to the cleanup call:

1. Find the Registry File with type equal to icb.dsm_rf_type and name equal to icb.dsm_rf_name. Otherwise, go to step 3.

2. If the UNLOAD disposal mode is specified in bdisp and if TEMPLINK is ON, then store blanks in the up(DOWNSLOT).uptype and up(DOWNSLOT).upname entries of the new RF.

3. Store zero in the in_use_sw in the RF.

4. Set TRACEDOWN equal to rf.att_types(rf.present_type_index).trace_down. Set LOOKONLY equal to rf.att_types(rf.present_type_index).look_only. Set DOWNTYPE equal to rf.att_types(rf.present_type_index).down_type and set DOWNNAME equal to rf.att_types(rf.present_type_index).down_name. Set TEMPLINK equal to rf.temp_link and set DOWNSLOT equal to rf.down_slot.

5. If the RELEASE disposal mode is specified in bdisp and if the allocate bit in the RF is ON, then make the following call to deallocate the device:

```
call de_alloc$resource(rf.devices(1).resource_name,cstatus);
```

If cstatus indicates that the device is no longer allocated to this user, go to step 11.

6. If the UNLOAD mode is specified in bdisp, attempt to delete this resource from the system by the following call:

```
call ioam$delete_resource(rf.alloc_type,
     rf.devices(1).resource_name,cstatus);
```

Ignore the value of cstatus.

7. If NDOWN is equal to 1, go to step 8. Otherwise, if NDOWN is less than 1, go to step 11. If it is greater than 1 and if the RELEASE disposal mode is specified in bdisp, then do the following for each of the NDOWN elements of DOWNNAME.

a. Find the RF with the given type and name.

b. If the allocate bit in the RF is ON, call de alloc$resource (see step 5 above).

Go to step 11.

8. If TRACEDOWN is OFF or if LOOKONLY is ON, go to step 11.

9. Find the RF with type DOWNTYPE and name DOWNNAME(1).

10. Go to step 2.

11. If a private DMP was created (i.e., icb.private_dmp is ON), call the central supervisor to destroy the DMP.

12. Return.

Summary of Cstatus Bits

1 Inaccessible or non-existent Registry File
2 Unlinked or improperly linked Registry File
3 Unavailable device
4 Attachment of DCM failed
5 No such type in att_types array of Registry File
6 No diverted path to revert
7 Attempt to revert or divert a presently diverted path
8 Validation level too high for trap quits or trap hangup calls
9 No Registry File with proper type found for get_rf call
10 Bad mode or disp
11 System bug
12 Bad argument