To: MTB Distribution

From: J. C. Whitmore

Date: 21 March 1978

Subject: IO Daemon Changes for MR 7.0

INTRODUCTION

This MTB describes enhancements to the IO Daemon and related user interfaces which are planned for release MR 7.0. Actually, many will be available in the 6.5 interim release. See the schedule on page 11. These enhancements have been requested in one form or another by many of the Multics sites. Some are linked to planned changes to the accounting mechanism and to the absentee facility.

The plan of this MTB is to define each problem we are trying to solve and to immediately present the initial approach to solving the problem. Some of the solutions are affected by previous solutions. These are identified where possible and their interactions are explained.

A. Changing The Priority of a Request

At many sites, operators are requested to run a certain dprint request immediately. Currently there is no mechanism to do this. The problem is similar to the need to run absentee job XX next or right now, but the number of request types (and therefore queues) makes the IO Daemon problem different.

This problem is actually three problems:

A1. Run request ABC immediately, i.e., stop the current request.

A2. Run request ABC next, i.e., to the head of queue 1.

A3. Run request ABC right after request XYZ, i.e., change the order in which requests will be run over some period of time.

Before the three problems can be attacked, we must solve a secondary problem: How can we uniquely identify a dprint request? This secondary problem comes from the use of long and short names for directories and from multiple requests for segments of the same name in the same or different directories.

No special solution is planned for problem A1. Since there are commands to kill and restart the current request, implementing a solution to A2 can also provide a solution to A1.

Problem A3 implies a command to move a request from one queue to the tail of another queue and possibly to change the order of requests in a given queue. The problem of doing the latter was discussed in detail by Casey in MTB-364 and will not be repeated here.

Design approach

The approach to handling each of the problems identified above is presented in the following paragraphs.

Identifying A Request

It is most often acceptable to identify a request by its entryname for a given user. In these cases it would be more convenient for the operator (and user) to list, cancel, etc. a request according to its entry name (a feature we don't have today.) Therefore a -entry control argument is proposed for the command descriptions for cancel_daemon_request, list_daemon_request and new commands as shown in Appendix II.

When there are multiple requests for the same entryname in different directories, the full path name is sufficient to identify a request (this is the only method available today.)

However, in some (albeit rare) instances, e.g., multiple requests for the same segment with different dprint options, another identifier is needed. Therefore, an optional -id control argument is proposed for list_daemon_request which will return a <match_id> for each request it lists. The <match_id> can then be used to cancel a request or for other uses as we shall see. The major issue here is choosing a mnemonic form of the <match_id> and defining the datum used for its generation. The various issues are discussed in Appendix I.

In practice, cdr and some new commands will also accept an optional -id control argument to define the <match_id> of the request to be cancelled, etc. The <match_id> will be used to resolve ambiguities in equal pathnames for the same user in a given request type and queue.

Run Request ABC Next

To solve problem A2 (since we are ignoring problem A1), the operator will be given a new driver command, "next_req", to tell the coordinator to run a specified request next. A complete command description will be found in Appendix II. This command must be a driver command so that operators at a remote RJE site can also use this feature. (Note: this introduces a requirement

for remote driver operators to be able to list requests in the driver's queues so they can identify the request to be run next. This secondary problem is covered under item "D" below.) The effects of the next_req command are as follows.

Any request promoted to the head of queue 1 by this command will be marked as coming from queue 1 for accounting purposes (coordinator will do this.)

A driver will only be able to ask the coordinator to promote requests which it can process, i.e., within its own request type and device class.

Any defered requests (see item "C" below) will be eligible to run next by this command.

The next_req command defines the request to be run next by the request_type.device_class, user id, segment entryname or pathname, and optional <match_id>. The coordinator will receive this request identification data through the coord_comm message segment and will search the request type queues starting with queue 1 until a request for the specified user and segment is found. This request will be the one given to the minor device driver the next time it asks for a request. If there is an ambiguity in the request identification, no errors will be reported. The first request which matches the criteria will be chosen to run next.

The coordinator should be able to stack several next_req command orders before the driver asks for another request to process. In any case, the operator could use step mode to control the exact order in which requests are processed (all at queue 1 prices, of course).

The driver must be changed to make this happen in correct sequence. Currently the driver has asked for the next request before it checks for a command. Now it will have to see if there is a command (possibly next_req ...) and process it before asking for the next request.

One problem remains with this approach: the coordinator has no way to tell the driver that it could not find the request. To solve this, a new ipc message will be defined for the coordinator to send back to the driver which indicates a positive yes or no. This must be done to keep the driver from asking for another request to process before a failed next_req command is recognized. (e.g., next_req -user jones -entry foo;go would cause the go to ask for another request even though the user Jones was misspelled.) This has the advantage that the driver can go back to command level and request a new (correctly spelled) next_req.

Run Request ABC after request XYZ

A partial solution to this problem is provided by the next_req driver command. However, the user will be charged dearly and must call the operator to get a higher priority. It would be better to allow both users as well as operations to move requests from one queue to another or to reorder requests within a queue (according to proper access controls of course.)

Therefore, a command to move a request from one queue to the tail of another queue will be provided. This command will use (or be part of) the command to move an absentee request from one queue to the tail of another queue. (At this time, no attempt will be made to reorder requests within a queue.) The new SSS command, move_daemon_request or mdr (with a subroutine interface), will use similar control arguments to those for ldr and cdr. A complete command description will be found in Appendix II.

Note on New Control Arguments

New control arguments, -id, -entry, and -user, will be provided for the list_daemon_request, cancel_daemon_request, move_daemon_request and next_req (driver) commands. The -id control argument to ldr will cause the <match_id> of each request listed to be printed (this is how we get the <match_id> values for cdr and mdr.) The -id control argument to cdr, mdr and next_req will input a <match_id> for selecting a request. The -user control argument to cdr and ldr will replace -admin. The function will be the same, but the name -user is more meaningful and will be the same as the arguments for mdr and next_req. The -admin control arg will continue to be supported for several releases for upward compatability.

When searching the queues for a request which matches user, pathname or entryname, and any specified <match_id>, the mdr and cdr commands will search from the end of the last queue toward the head of queue 1 until a request which matches the criteria is found. Ambiguities will not be diagnosed (i.e., if two requests would match, the first found will be used.) However, as mentioned above, the next_req driver command will search the queues from the front back to find the next request to run.

B. Forms Accounting

The IO Daemon released with MR 6.1 contained variable page size and slew to channel features which are very useful in producing output on preprinted forms. Several sites have requested the ability to charge different rates for using special form stock. Currently the IO Daemon prices are fixed per queue for all request types. We need a way to charge different rates for each queue of a request type.

Design Approach

There is a planned extension to the accounting mechanism which would allow the site to define certain "resources" and set a price for each. The details of this plan are beyond the scope of this MTB, but the proposed user interface will be described briefly.

A new entry to system_info_ (get_resource_price) will be added which will take an ascii resource name as input and return the current price or an error code. The resource price list will be some form of a linked list of name-price pairs (something like the value_seg). An administrative command will be used to set or change the price of any resource the site defines. (Accounting extentions to provide a breakdown of accumulated charges by resource for each user are also planned.)

With this site defined price list and subroutine interface, the administrator can define a resource for the line charge for each request type and queue (or even make them all the same by only defining one IO price).

The binding of a resource price to a request type and queue will be done using the iod_tables. A new substatement, line_charge, for request type entries will be defined which will look like:

```
Request_type:       printer;
 generic_type:      printer;
 line_charge:       prt_q1, prt_q2, prt_q3, prt_q4;
 device:            prta;
```

There will be one required entry for each queue defined for the request type. Each string, e.g. prt_q3, will define the line charge resource price when it is passed on to system_info_$get_resource_price. The iod_tables compiler will check to see that there are the correct number of prices defined for each request type and that each price is defined in the system price list. The actual resource price string will be stored in the iod_tables for access by the driver during initialization. This will allow a price change to take effect without recompiling the iod_tables.

During driver initialization, iodd_ will pass a pointer to the resource price information of the iod_tables to io_daemon_account_$init (a new entry). The prices for each queue will be obtained from system_info_ and stored in internal static. If a price is undefined, the driver will fail initialization. This approach does not preclude other pricing strategies for the future, e.g., per page charges for any form type, cpu charges or real time charges.

As each request is processed, io_daemon_account_ will be called in the usual way to compute the charge for each copy finished

using the stored prices. Notice that the prices will be constant for the life of the driver, but are re-established whenever the driver switches to a new_device or re-initializes. Currently, the queue prices are fixed for the life of the process. We could make the price changes more dynamic, i.e., call system_info_ for the prices each time io_daemon_account_ is called, however, IO prices are changed so infrequently, that the extra overhead is not justified.

For now, all IO Daemon charges accumulated by a user will be stored in the PDT just as they have been for some time. This means that the total IO charge will no longer be subject to verification by multiplying the total lines per queue by the IO rate for the queue. This problem will be solved in later enhancements to the accounting mechanism.

C. Do Certain Requests Later

Several sites have requested the ability to place certain requests into a form of "defered state" to be processed at some later time. Sometimes the decision to defer a request would be based on attributes of the request, like how long it will take to run. Therefore, an automatic defer mechanism would be useful. At other times, an operator interface to defer a request would be useful.

Design Approach

For the simple case of defering a request which has not yet been seen by the coordinator, we will allow the move_daemon_request command to be the solution. The only problem with this is that the request will lose its position in the queue. But, it will definitely reappear automatically at some later time without further action by the operator.

For the other cases, the driver needs a mechanism to tell the coordinator that it has bypassed its current request. The existing "keep_in_queue" bit in the request descriptor is meant for this purpose. On receipt, the coordinator will not delete the request from the queue, but will rewrite the request (preserving the sender info in the message header) marking the request as being in the defered state. This will allow the user to list the status of his requests and see that it is not being run even though it is at the head of the queue. The coordinator will continue passing requests to the driver starting with the next request in the queue.

To complete the mechanism, we need to define a new ipc message from the driver to the coordinator which will cause the coordinator to go back to the head of each queue for the request type of the requesting driver. This new ipc message will be sent to the coordinator on driver command only. (See the restart_q

driver command in Appendix II.) This will allow remote RJE sites to go back and pick up any defered requests. (A future extension might be to have the coordinator automatically go back to the head of the queues when there are no more requests to run. This will require more thought to avoid looping due to automatic defering of requests.)

There are advantages of this approach over a special "holding" queue (as some have suggested.) The requests will retain their place in the queues and actually get first priority when the coordinator goes back to the head of the queues. Defered requests will automatically be re-examined when the coordinator is reinitialized (generally once a day). Also, the need to have one "holding" queue per request type is avoided, thus saving extra seldom used segments.

With this basic mechanism, we can implement both an automatic defering of requests which exceed some processing criteria and an operator command to defer the current request (probably given after a QUIT).

The initial criteria for automaticly defering a request is: the request will take too long to process, and the line length requested is too long for this device.

To provide the first automatic criteria, we need a way to estimate the time needed to process a request. The estimate does not have to be too accurate. The request processing program (e.g. do_prt_request_) will maintain an exponentially smoothed average of the number of bits per second of real time (call this Rate) used for IO operations. Then,

time_estimate = n_copies * bitcount / Rate

A driver specific command will be provided to allow the operator to specify the maximum time he will allow for a request. (See set_defer_time in Appendix II.) For each new request, if the time_estimate > defer_time, the request will be defered. The time_estimate can also be used in the log message to warn the operator that the current request will take <M> minutes to process.

For completeness, Rate will be reset during driver initialization. The current processing time will not be added into the Rate calculation if any errors occured or if any conditions were signaled. The first request will always be processed and its time will be the first value of Rate. Values outside of 20% of Rate will be discarded. Separate Rate values will be needed for each stream/switch in output_request_ e.g. print and punch.

For the line length criteria, the user's requested line length will be compared to the phys_line_length of the device and the

request defered if there is not enough space. Therefore, users who really don't want their requests spooled onto a 132 character per line printer will be saved.

For any request defered automatically, a message to the operator will be printed (and maybe the user will be notified if the notify bit is on).

The operator will be able to manually defer a request by issuing the new standard driver command "defer" (See command description in Appendix II.) This command can be given at either request command level or at quit command level. The effect will be the same as though the request had been defered automatically.

For the first implementation, any defered request will eventually be run from its original queue for accounting purposes. Later, we may want to think about giving the user a rebate for delaying his request.

The list_daemon_request command will be changed to ignore defered requests, unless requested by the new -defer control argument.

D. Site Defined Operator Commands

There seems to be a growing need to provide the ability for RJE terminal operators to execute normal Multics commands which are not known to the IO Daemon driver list of closed sub-system commands. The most notable is the need to see if there is anything in the driver's request type queues. Next is the need to list RJE jobs submitted by the remote station. Then comes car, move_abs_request, run job ABC now, cdr, ldr, ... and the list goes on.

Design Approach

Clearly we cannot add code to iodd_command_processor_ every time some new bell or whistle is added to the system. What we really need is a form of "x command" for the operator to be able to execute a site defined exec_com, just like the initializer x command.

The problem here is one of the structure of the IO Daemon. Each of these commands, ldr, lar, etc., has been written with the assumption that it can print on user_output, or error_output and talk to the user. At a remote RJE station, the operator receives output over some other stream (switch) while user_output is connected to the initializer's console. Imagine the output from ldr appearing on the operator's console without being requested and the poor RJE operator seeing nothing.

Therefore, an x command for the driver is proposed which will reattach the user_output stream (switch) to the slave output

switch (with a proper any_other handler of course) and call exec_com with the remainder of the command line. An attempt will be made to allow the operator to respond to questions asked by a command, however, due to the strange nature of input and output for remote devices, this may not be practical. All conditions raised which would terminate an exec_com will be fatal errors to the x command. The exec_com will be called iod_admin.ec and will be stored in >ddd>idd, the IO Daemon root directory.

A special active function for drivers will have to be defined so the iod_admin.ec can get the names of the request type and device if the driver calling it. This is needed so that a simple "x list" can be used to list the requests in the queue for a particular driver without using up the first several exec_com arguments (which would not be extensible.)

E. Positive Binding of RQTI Segment to a Request Type

The original MCR which added request type info segments (rqti segs) to the IO Daemon stated that a new keyword would be added to the iod_tables for each request type entry so that the name (and need for) a rqti seg would be bound to the request type.

The initial implementation simply initiated a segment in a particular directory. If the initiation failed, default values were used. It is possible for the rqti seg to get deleted, damaged, etc. and its need for correct operation of the request type would not be noticed until some requests were improperly printed (and possibly deleted).

Design Approach

This change is trivial while the iod_tables compiler is open for changes to support the items above. The original plan should be followed.

The syntax of the new iod_tables entry would be:

```
rqti_seg:          printer_info;
```

The ascii string "printer_info" would be stored in the iod_tables. The driver would find this entry for its request type. If the string is non blank, the driver will initiate the segment in >ddd>idd>rqt_info_segs. If initiation fails, initialization of the driver will fail. The keyword would be optional and if not given, a blank string would be stored.

F. Max Queues per Request Type

Sites which use many request types for special forms find that one queue will generally be sufficiect while 4 queues are often needed for the normal printer request type. In cases where 4 queues are needed for any request type, the current implementation requires 4 queues for every request type (or possibly some tricks like an empty dummy queue with all the added names of the unused queues). It seems nunecessary for sites to maintain extra queues or to use tricks to get around our software limitations.

Design Approach

Add a max_queues substatement to each request type statement in the iod_tables. This keyword would be optional. The default would be given by the global Max_queues statement.

The iod_tables compiler would place the number of queues in the request_type entry of the compiled iod_tables. The print_iod_tables command would display any value not equal to the global value.

The create_daemon_queues command would use the per request type value for the number of queues to create, instead of the global value.

The coordinator would only look for queues 1 to max_queues.

With the new accounting mechanism described under problem B above, there is no longer a problem defining a default queue other than queue 3 due to fixed pricing. Thereand default_queue fore, iod_info_ will be changed to return the max_queues and default_queue for each request type. The commands dprint, ldr and cdr will use these values. The default queue will be defined as the highest numbered queue or queue 3 if there are 4 queues. This allows queue 4 to become a low priority queue. (An alternative would be to define the default in the iod_tables. But this has not been requested by anyone and may not be worth the trouble.)

G. Don't Delete User Segment If Error Occurs.

Several errors are currently hidden from the driver by the device dims. These include: out_of_paper, paper_low, manual_halt, etc. For printing and punching, the user usually wants a nice clean listing or card deck. However, these conditions may have prevented this. The user should get another chance to issue his request if operatiors fails to use the restart command after these conditions occur.

Design Approach

The printer dim will be changed to record the number of errors which occur while printing. This will be stored in the SDB. A new order, "error_count", will be defined, which will return the current error count value. The error count value will be set to zero by the "reset" order.

The do_prt_request_ program will be changed to make the "error_count" order call after each copy of the request has been printed. If the count is greater than zero, the dont_delete flag will be set to prevent the coordinator from deleting the segment, if the user has so requested. When the deletion has been cancelled in this manner, the user will be notified if the notify flag was set in the dprint message.

A similar change will be made to output_request_. However, the "error_count" order call will result in the current non-action if the undefined_order_request code is returned. This will allow the "error_count" order to be added to the card dims as time permits.

Schedule For Implementation

For the interim MR 6.5 system (about June 1978) the following problems should be fixed:

A. Changing The Priority of a Request.

B. Forms Accounting.

D. Site Defined Operator Commands.
(Without operator response to command query)

E. Positive Binding of RQTI Seg to Request Type.

The remaining tasks will be completed by MR 7.0:

C. Do Certain Requests Later.

D. Site Defined Operator Commands.
(With operator response to commands, if possible)

F. Max Queues per Request Type.

G. Don't Delete User Seg on Errors.

Appendix I

Choosing A dprint Match Id

Choosing the <match_id> for identifying a dprint request seems to be the most controversial subject of the user interfaces described in this MTB. I will wait for the design review to resolve the final approach. The major issues are:

1. Should the <match_id> be unique among all queues of all request types? Or, just within a request type? Or, just within a given queue? Or, just within the process?

2. Should the <match_id> stay with the request if it is moved from one queue to another? To another request type? I If it is moved by either the owner or by the operator for the owner?

3. Should the <match_id> be a small character string? A clock time number? Part of a clock time for ease of typing? A dprint job number like absentee job numbers?

4. Should the <match_id> be printed on the head_sheet? In the log? Is it really part of the request?

To keep this all in perspective, the user and pathname will uniquely identify most all requests. We need the <match_id> to resolve ambiguities only in rare cases. But, if the <match_id> is truly a request unique id, the need for typing the request type, queue, segment name and possibly user name can be avoided. Some possible approaches are described here to stimulate discussion.

One approach would be to assign unique dprint request ids (like the proposed absentee job ids) as each request is put into a queue. This would have the advantage that only a single request id would be needed to identify a request in any of the queues of any of the request types. However, there are problems. Rewriting each message by the coordinator (which would assign the ids) would introduce additional overhead, due to the frequency of dprints. A finite list of request ids are desireable to keep the mnemonic small (who likes !BBBxRQhdPkBnT.) However, some ids can remain in the queues indefinitely, and use up available ids. In addition, when a request is cancelled by the user or mseg salvager, the coordinator may not be told to reclaim the request id and therefore must verify all outstanding ids during initialization by reading each request in each queue of each request type.

Anotner approach would be to form the <match_id> from the time the request was added to the queue. This time can be found in the request header. (The message id could be used if we wanted to depend on it's being a true time. However, the match id would

then have to change if a request was moved from one queue to another.) The time data, being part of the request, cannot be assumed unique and can be set to any value by any process which can rewrite the message (not secure). The mnemonic form of the match id could be YYMMDD.hhmmdec (e.g., 780316.1625243210 for 3/16/78 4:25.2... pm). Not all the string would be needed to match against the request <match_id>. The following strings would match the above example: 1625, 5.16, 316., 16.1625 and etc. The rule is: If no "." is given, the number is taken as the right hand part of the <match_id>. If a "." is given, the digits before and after the "." are taken as the right and left parts of the <match_id>. The digits specified for the right and left hand parts must be equal to the same digits in the <match_id> for the match to occur.

A third approach might be to add a 36 bit unique fs_time to the message header. This message unique id could stay with the request as long as it remains within ring 1. A move message primitive for moving a request from one queue to another could then preserve the id for the move_daemon_request command. The mnemonic form could be the same as above, but shorter maybe. Again, this message id would be unique among all requests in ring 1. The drawback is the size of the task of changing the message segment format, converting old versions, and salvaging.

If the <match_id> was not going to be used to tell the operator which request to run next, a simple assignment of short per process character strings associated with request ids (prepared by ldr) would do. But then, would it be worth doing at all?

Appendix II

Command Descriptions and Documation Requirements

Standard User Commands (MPM Commands)

Name: move_daemon_request, mdr

The move_daemon_request_command is used to move a request from one IO Daemon queue to another. The move may be within the same request type or from one request type to another. The request will always be placed at the end of the target queue.

Usage: move_daemon_request path -control_args

Where:

path is the segment pathname. (Required.)

The following control arguments may be used:

-entry specifies that the request will be identified by the entryname portion of the pathname. (Optional.)

-id <match_id> specifies that the request selected must also have the specified match id. (Optional.)

-rqt <A> specifies that the request to be moved will be found in request type <A>. If not specified, request type printer will be assumed. (Optional.)

-q N specifies that queue N of the request type contains the original request. If not specified, the default queue will be assumed. (Optional.)

-to_rqt <B> specifies that the request should be moved to request type <B>. If not specified, the original request type will be used. Request types <A> and <B> must be of the same generic type. (Optional.)

-to_q N specifies which queue to move the request to. (Required.)

-user specifies the person (optionally person.proj) of the submitter of the request to be moved. The default is the group id of the process. This control arg is primarily for the operator. Both r and d extended access are required. This control argument will cause the command to attempt to use privileged mseg primitives. (If the user lacks access, standard

primitives will be used.) The special mseg primitives will preserve the original identity of the submitter. The AIM ring_1 privilege will be needed to preserve the original AIM attributes.

Name: list_daemon_request, ldr

The list_daemon_request command .... (same as MPM)

New optional control arguments:

-id specifies that a match id should be printed for each request that the command lists. This match id can be used with the cancel_daemon_request command.

-entry <entryname> specifies that the requests listed should have the given entryname.
(**** Only if time permits. ****)

-user <name> specifies that only requests for the specified user (person or person.project) should be listed.

-defer specifies that only defered requests should be listed.

Name: cancel_daemon_request, cdr

The cancel_daemon_request command .... (same as MPM)

New optional control arguments:

-entry specifies that only the entry name portion of the path argument will be used to select the request to cancel, starting from the end on the queue.

-id <match_id> specifies that the request to be cancelled must have a match id which corresponds to <match_id>.

-user <name> specifies that the request to be cancelled must have been submitted by the given user (person or person.project). The default is the group id of the process. (This control argument requires special access not given to most users.)

Name: temp_arg

The temp_arg active function is used to return character strings previously defined by the process. It is similar to the value active function. The values are stored in a temporary segment in the process directory.

Usage: [temp_arg Key]

Where Key is the name of the character string whose value is to be returned. Key may be up to 32 characters long and the value returned may be up to 168 characters long. The value of Key must have been previously defined in the process, otherwise the string "undefined!" will be returned.

The temp_arg active function is used by the IO Daemon driver to pass along certain names to the iod_admin.ec. The standard reserved Key names for the IO Daemon are: request_type and major_device.

The value of Key is defined by calling temp_arg$set:

```
Usage:    declare temp_arg$set entry (char (*), char (*));

          call temp_arg$set (Key, key_value);

  or from command level

          temp_arg$set  Key  key_value
```

IO Daemon Driver Commands (Multics Bulk I/O Manual)

Name: next_req

The next_req standard driver command is used to specify which request is to be run next. It may be issued at any driver command level. If a request is in progress, it will be completed (unless killed, cancelled or defered) before the specified request will be run.

Usage: next_req {dev} path -user <name> {-id <xx>}

Where:

dev is the optional minor device. It is required for a driver with multiple minor devices. It serves to identify the request type and device class for the

driver.

path is the segment pathname. (Required.)

-entry specifies that only the entryname portion of path is to be used to find the next request. (Optional.)

-user <name> specifies the name of the submitter of the request. Generally the person_id alone will suffice, but person.project will be accepted. (Required.)

-id <xx> specifies that the request to run next must have a match id which corresponds to <xx>. (Optional.)

Name: restart_q

The restart_q standard driver command is used to tell the cordinator to go back to the head of all the queues for the request type of the driver. This is done when some requests have been defered and are now to be run.

Usage: restart_q {dev}

Where:

dev is the optional minor device name. It is required for a driver with multiple minor devices. It serves to identify the request type of the requesting driver.

Name: defer

The defer standard driver command is used to defer the processing of the current request. It is used after a QUIT (like the kill and cancel commands), but the request will remain in the queue to be processed later (See the restart_q command). Any defered requests will be reprocessed whenever the coordinator is initialized.

Usage: defer

Name: set_defer_time

The set_defer_time device specific driver command is used to set the time limit for automatically defering a request. Any request received by the driver which has an estimated processing time greater than the defer time will be defered for processing at a later time.

Usage: set_defer_time {<time>}

Where <time> is the value of defer time to be set. The <time> is specified in minutes and applies to all copies of a request. If the <time> argument is not specified, the current defer time will be printed.

Name: x

The x standard driver command is used to allow each site to define any special commands for the driver as needed.

Usage: x <command> <args>

Each <command> string is taken as belonging to a label in the iod_admin.ec. If a label is not found, an error message is printed. The site administrator will create the iod_admin.ec with any labels he chooses. Each label will correspond to a set of commands which may be executed by the driver process. Special keywords to the temp_arg active function (request_type and major_device) can be used to specialize the iod_admin.ec to the needs of the driver.

All <args> will be passed on to the exec_com in the form of &1 &2, etc.

Changes to IOD Tables (Multics Bulk I/O Manual)

The following substatements must be added to the descriptions of the "Request_type:" statement.

line_charge: P_q1, P_q2, P_q3, P_q4;

The line_charge substatement defines the resource prices for the line charge of each queue of the request type. This substatement is required. The resource names for each queue must be given in order starting with queue 1.

max_queues: N;

The max_queues substatement is used to refine the value specified by the Max_queues statement to apply to an individual request type when it is different. The value of N may be from 1 to 4.

rqti_seg: entryname;

The rqti_seg substatement is used to define the name of the rqti segment to be used with the Request_type. This substatement is optional. When specified, the entryname must correspond to a segment entryname in the >ddd>idd>rqt_info_segs dieectory. If the segment does not exist, an error message will be printed. When not specified, no driver will look for a rqti segment for this Request_type.

Supporting Documentation

The changes described in this MTB assume certain changes to related procedures which are not being implemented by this set of tasks. These are:

system_info_$get_resource_price (resource, price, code);

ed_installation_parms

Proper documentation of these interfaces will be verified as this MTB is implemented.