To:	Distribution
From:	Richard J.C. Kissel
Date:	10/20/81
Subject:	Design of a General Interface and Implementation Structure For Use in a Networking Environment

Send comments to:

Kissel.Multics @ MIT
Kissel.Multics @ System M

or

Richard J.C. Kissel
Honeywell Information Systems
575 Tech. Sq.
Cambridge, Ma. 02139

or

Mail stop: MA22

or call:

617-492-9319
HVN 261-9319

INTRODUCTION

This MTB defines and examines the issues relevant to the design of a general user and system interface that can be used in a networking environment. A specific design is also proposed. This design uses many of the concepts and much of the terminology developed in a draft MTB by M.R. Jordan in response to the requirements from CNO for Level 6 attached processor support. The general problem can be stated as follows: given the fact that Multics is going to be connected to a variety of different networks, each providing some common and some unique functions, and each connected to possibly intersecting sets of remote systems, how can we provide users with a unified and extensible interface which will make these diverse capabilities easy to use?

This MTB provides a framework and the necessary terminology with which to discuss network applications. The framework divides an application into a part concerned with queueing network requests and managing the request queues (handled by system code), and a part which is application specific (handled by application code). The interface that the application specific part must have is also specified. This is the Network Application Support Procedure (NASP) interface.

In the rest of this MTB I will use the terms "local" or "local system" to refer to the system that a user (or absentee job) is logged in to. The terms "remote" or "remote system" will refer to any system which is accessible from the local system through a network. Examples of networks with which Multics will be concerned are: ARPANET/INTERNET, DSA, X25, a hyperchannel connection, local networks, and networks accessed through dialout capabilities.

FEATURES OF A USER INTERFACE

In a networking environment, a user is generally concerned about three things.

The first is the function he wishes to perform remotely. This includes functions such as: file transfer, remote job submission, mail, etc. The user is always concerned with, and must specify explicitly, the function he wishes to perform. Notice that remote login (e.g. as done by the current telnet command) is just another network function that fits into the interface defined here.

The second thing a user is concerned about is the remote system at which the function is to be performed. The user may or may not be concerned with the specific remote system that will perform the function. For instance, a file transfer will require that the user specify the remote system, because only the user knows where the file currently is, or where it should be created. On the other hand, for a remote job submission, the user may only need a remote system with certain attributes (e.g. a Level 6 running GCOS6 MOD 400 and FORTRAN release 2) rather than a specific remote system. In some cases it may be possible to deduce the remote system attributes required from the function to be performed. This would relieve the user from any concern for the remote system.

Finally, the user is concerned about the type of network through which he reaches a remote system. In many cases there may only be one type of network connecting the local and remote systems. In other cases there will be multiple network types (e.g. ARPANET, DSA, X25, etc.). In general, the user does not care what network is used to reach a remote system to perform a specified function. However, in some cases the user may want to use special knowledge (e.g. relative speeds of various networks, the fact that the ARPANET is only supposed to be used for ARPA related traffic, etc.) to determine which network will be used. Also, some functions may only be available on a specific network. In this case, however, the system should be able to relieve the user of worrying about which network to use by picking the correct one itself.

The foregoing discussion should make it clear that the user interface in a general networking environment must provide certain features. The user must explicitly specify the function he wishes to perform on a remote system, together with any information necessary to perform that specific function. The user must be able to specify a particular remote system or specify the attributes that a remote system must have to perform the requested function. And, the user must be able to specify a particular type of network over which the request to perform the function will be sent.

In addition, one more feature for the user interface is necessary on Multics. That is, a way to distinguish between a request that is to happen in the user's process (while he waits) and a request that is to be queued for later processing by a daemon, with the results reported back to the user. If the request is to be queued, the user must provide information to allow the queue management software on the local system to handle the request. Since there are a number of commands on Multics which manage queued requests in a consistent manner, the user interface for network requests should match these other interfaces as closely as possible (i.e. similar control arguments, and the ability to enter, list, cancel, and move requests).

A user command interface which embodies the features discussed above is proposed in Appendix A. A subroutine interface will be provided for use by commands which currently perform functions locally but which could be extended to perform the same function remotely (e.g. send_mail).

USER LEVEL IMPLEMENTATION ISSUES

The functions that users might wish to perform are extremely varied. Some are well known, for instance, file transfer and mail. Others are very user and system specific, for instance, "send this FORTRAN program to a Level 6, compile it, execute it, and return the listing and execution output to me". Also, functions that a user wishes to perform will depend on the types of network connections available on his local system, and the other systems reachable through these networks. Support of these varied functions requires a design that allows Honeywell to provide some functions, but also allows users to write their own functions.

The proposed solution to this design problem is the concept of a Network Application Support Procedure (NASP). A NASP is a Honeywell supplied or user supplied piece of code which performs a desired end user function. A NASP must perform two basic actions to implement and end user function.

The first action is to take command line arguments specified by a user and generate an application specific structure containing the information necessary for the execution of the function. For instance, a file transfer NASP would parse command arguments describing the files to be transferred, and include the local and remote file pathnames, and the hosts on which they reside, in the application specific structure.

The second action is to take the structure generated by the first action and perform the requested function. This includes interfacing with the appropriate network software required to reach a specific host and possibly interacting with the user or operator.

The reason for this partitioning of NASP actions is that the application specific structure may either be used to execute the function in the user's process, or it may be placed in a queue for later execution in a daemon process. By placing all of the application specific knowledge in the NASP, it will be possible to have a single set of general purpose queue management software for many different applications.

One NASP will be required for each different user application function to be performed. The name of the NASP procedure to call is found by using the Network Information Table (described below) to map the function name to the NASP name. This name will then be used with the "network" search list to find the subroutine which implements the specified function. Each NASP will have standard set of entries defined to perform the various NASP actions. A NASP can be written to perform a function over a number of possible network connections, but this will usually be done by separate subroutines in the NASP, since different networks have different interfaces and require different protocols to perform similar functions. In fact, a NASP may have a set of internal interfaces for performing network specific actions similar to the defined external NASP interfaces. Thus, the NASP entries would just perform a routing function based on a user supplied (or internally generated) network name. The internal subroutines would then perform the function on the specified network. Although not currently defined, these internal entries might be found by using the Network Information Table to map a function name and network name to a set of entries to perform the function on that network.

In the proposed design, a clear distinction is made between queue management and the processing required to carry out a given function. All processing related to the queuing command arguments, putting requests into queues, taking requests out of queues, and calling the appropriate NASP entries, is handled by system supplied software. A standard request header is defined for this purpose (see Appendix D). All processing related to handling a specific function is done by user or system supplied NASP entries. Uninterpreted space is provided in a request for this purpose. The actual names and calling sequences for NASP entries are detailed in Appendix B.

A standard state for each request in a request queue is maintained in the standard request header by the queue management software. If more application specific state information is needed, it will be kept in the applicaiton specific part of a request and maintained by the NASP. This standard state has the following values and meanings.

The "deferred" state is set if the request's starting time (as specified by the user) has not yet been reached, or if the request has been deferred indefinitely (awaiting release by an operator or the user), or if the request is being held pending the completion of some other request(s). A request in this state will only be looked at by the queue management software to change its state or to update other queue specific parameters. It may also be passed to the NASP_$list or NASP_$modify entries to have application specific information listed or modified.

The "ready" state is entered when the request is no longer deferred for any reason. A request in this state will be given to a NASP_$execute entry to be performed whenever it is found by a daemon process looking for work. While the NASP_$execute entry is executing the request will be locked in the queue, and thus, unavailable to any other process for execution. The NASP_$execute entry will return with an indication of whether the request is "complete" or "not complete", and an indication as to whether any requests being held for the completion of this request should be released. The reason for having the two indications is that, normally, a request that is "complete" will cause the release of any related requests being held, and a request that is "not complete" will not. However, in the case of a "complete" request, the NASP may have notified the user of some exceptional condition, and therefore, not want to release any held requests. For instance, the request may abort in some way which makes retrying it impossible. And in the case of a "not complete" request, the NASP may still want any held request to be released. For instance, a file transfer NASP might use the "not complete" state to allow a grace time to expire before deleting the source file used in the transfer, but since the transfer has actually completed, any held requests can be released.

If the request is "complete", the queue management software will delete the request from the queue. If the request is "not complete", the queue management software will update the request in the request queue, making its state "not complete", and unlocking it so it may be processed later.

A request in the "not complete" state is handled in the same way as one in the "ready" state. But notice that the NASP must be carefully written to be able to pick up a partially completed request and continue to process it.

The user will be able to identify a queued request (in order to list, modify, or cancel it) by using a function name and request_id combination. The request_id will be a standard Multics request_id as described int he MPM Reference Guide. The function name will determine in which set of priority queues the request can be found, and the request_id will identify the particular request in those queues.

Network Information Table

There is a body of generally useful information which must be kept about the networking environment on a system. This information will be kept in the Network Information Table (NIT) on Multics. It contains the following general classes of information: a mapping from application function name to NASP name; a mapping from function name and priority to the name of a message segment to be used as a request queue; and a table of information concerning all the network connections available from the local host. This table lists: each host by name along with any additional names it may have; which networks may be used to access that host; the network address of that host on each network; the functions supported by each host and network connection; and the attributes of each host and each network. This list is not exhaustive and may be extended as we gain more experience.

Since this table must be updated and accessed in very general ways, it will be implemented as a MRDS database. Standard interfaces for entering, updating, and retrieving information in the database are detailed in Appendix C. A user who wishes to access the database in some way different from that provided by the standard interfaces is free to use the database directly through MRDS interfaces. It is expected that a NASP will need information from the Network Information Table to do its work (e.g. host attributes, function specific addresses, etc.).

DAEMON LEVEL AND QUEUE MANAGEMENT IMPLEMENTATION ISSUES

The following section addresses the issues involved in managing queued requests and constructing a framework within which daemon processes which handle queued requests can operate.

Queue Management Implementation Issues

The queues in which requests are put will be implemented using message segments. There will be one set of priority queues per defined function. An interface will be provided to enter requests into a specified queue. This will be used by the network request commands in the user's process to enter requests containing the standard header and the application data returned from a NASP_$parser entry. An interface will also be provided to extract requests from the queues based on various selection criteria. This capability will require an additional message segment primitive to support it, since requests may be pulled from the queue, examined, and put back unprocessed. This primitive will perform a "read and lock" operation on a message in a message segment (see the forthcoming MTB on message segment enhancements). Given this additional message segment primitive, no "coordinator" process will be needed to manage the queues. Instead, each daemon process will be able to get requests to process directly from the queues, using the interface specified in Appendix D.

When a call is made to get a request from a queue, the queue will be searched for a request which matches the specified criteria and is in the "ready" or "not complete" state. Requests which are locked will be skipped. Each daemon will have the option of searching the queue from the beginning or starting from the last request seen. Normally, a daemon will scan the queue, with each call starting from the last request seen, until it finds one it can process. It will process that request, and then scan the queue, with the first call starting from the beginning of the queue, for the next request to process.

Daemon Process Implementation Issues

All of the daemon processes which handle requests from the queues do so in a fairly standard manner. The daemon calls a queue management interface with some appropriate selection criteria to get a request to process. The daemon then calls a NASP_$execute entry with the request. The NASP_$execute entry does the actual work, just as in the interactive case, and returns with the indications described above. The daemon does any queue management operations required and then gets another request to process and the cycle continues. In order to facilitate administrative and operator control over the daemon processes, a standard start_up.ec which puts the daemon into the subsystem utilities environment will be used. A standard set of commands will be defined and controlled with the subsystem utility subroutines. This will allow the operator to communicate in a standard way with a daemon process about anything related to the queues or the selection criteria the daemon uses to get requests. It is still possible that a NASP will enter into some sort of question and answer dialogue with the operator if that is required to process a specific request. The specific features that will be provided in the daemon process environment are detailed in Appendix E.

Server Daemon Implementation Issues

In a networking environment there is a need to provide for "server" processes to handle incoming network requests. For instance, in DSA a file transfer server is necessary to handle incoming requests for file transfers to or from the local host. The primary difficulty in implementing a server is providing for security on the local system. The ideas that will be used here are: the "pool" storage specified in MTB 425 and MTR 158 for incoming data, and an extension of the concept of the "card input password" to a "network request password" which can be used to verify remote users.

Administrative Issues

In the above design, the basic objects which must be administered are: the NIT, the request queues, the requestor daemon processes, the input "pool" storage, and the extended "network request password". The administrative interfaces to be used to control each of these objects is detailed with the specific descriptions of each object in the appendices. Also, Appendix F provides a preliminary MAM type description of these interfaces.

IMPLEMENTATION PLANS

In order to gain experience with this design, we need to have an application in mind for the first implementation. That application will be a network independent file transfer facility. We currently have, or plan, file transfer capabilities over three types of networks: the ARPANET, a direct or dial_up connection (using bisync or some other line protocol), and DSA. One of these networks, either the direct connection or DSA, will be chosen for the initial implementation. The application specific control arguments that the user may specify are detailed in Appendix G (i.e. the arguments that may be given after the -arguments control argument). Even this first implementation will require implementing all of the framework described above. That is, the user commands, the NIT (at least some portion of it), the queue management software, the daemon command environment, and server processes; as well as the actual NASP software for the file transfer application.

IMPLEMENATION CONSIDERATIONS

The framework described above will be part of the standard system. This includes the user commands, the NIT, the queue management software, the daemon command environment, and the server processes. The MRDS database for the NIT can be shipped as an unpopulated database, so the only MRDS facilities needed in the standard system are the ones which allow database inquiry and update. Specifically, no MRDS facilities for database creation will be needed in the standard system. It is hoped that this restriction will allow the necessary parts of MRDS to be shipped as part of the standard system. It is also intended that the entire framework can be shipped separately from any standard release. Each NASP supplied by Honeywell can be a PSP, in particular the file transfer NASP described as part of the initial implementation can be a PSP.

The initial implementation as described above should take about 6 man months for the framework and 3 man months for the

NASP. These two tasks can be done concurrently, so the total calendar time should be about 6 months.

BASIC SCENARIOS

In order to illustrate how the defined interfaces will be used, three scenarios for a file transfer application are given below.

Interactive Scenario

Assume the user types the following command:

```
nr ft foo.pl1 bar.pl1 -at Mit-Multics
```

The network_request command does the following basic things:

- o Calls nit_$get_nasp_names with the function name "ft" to get the name of the NASP which implements the requested function. Assume there is a NASP named file_transfer_ which does file transfers. The name "file_transfer_" is then used with the "network" search list to find the actual subroutine to call.

- o Calls file_transfer_$parser with the rest of the command line after the "ft". This entry returns a file transfer specific data structure.

- o Calls file_transfer_$execute with the data structure obtained from the parser. During execution, this entry will have to call various nit_ entries to get information to carry out the transfer, e.g. what networks the host "MIT-Multics" can be reached through, the address of "MIT-Multics" on a particular network, and perhaps, the name of the file transfer server on "MIT-Multics". This entry will then carry out the file transfer using some appropriate protocol and print out any necessary information for the user. It then returns.

- o The network_request command then returns to the user.

Queued Scenario

Assume the user types the following command:

```
enr ft foo.pl1 bar.pl1 -at MIT-Multics
```

The enter_network_request command does the following basic things:

- o Calls nit_$get_nasp_name as described above.
- o Calls file_transfer_$parser as described above.
- o Checks any queueing control arguments given (there are none in this example) and sets any queuing defaults.
- o Builds a network_request structure by filling in the standard header, and putting the data structure from the parser at the end. It then calls nrq_manager_$put with the request structure to put the request in an appropriate queue.
- o Prints the request_id returned by nrq_manager_$put, and returns to the user.

Daemon Scenario

Assume the daemon has been started and the "go" command was issued. Then the following basic steps occur.

- o A call is made to nrq_manager_$get with a function that the daemon is supposed to handle. It calls nit_$get_nasp_name to get the NASP to call (as described above). Assume it is the file_transfer_ subroutine (as described above). The daemon checks to see if the host and network specified by the request are ones that it can handle by calling file_transfer_$info. If it cannot handle this request it puts it back in the queue and gets another one. If it can handle the request then it continues as follows.
- o The file_transfer_$execute entry is called with the application information from the request. This entry performs the actions for the file transfer as described in the interactive scenario.
- o Assuming the file transfer was successfully completed, the daemon calls nrq_manager_$delete with the request and updates any requests being held for completion of this request. The daemon then calls nrq_manager_$get to get another request and continues the cycle.

APPENDIX A

This appendix gives MPM style documentation for the user command interface being proposed for general networking requests.

Name: enter_network_request, enr

The enter_network_request command is used to queue a network application request for later execution. A request id is printed by this command for use in later enter_network_request, cancel_network_request, list_network_request, and modify_network_request commands. The request id is a standard Multics request id as described in the MPM Reference Guide. See the network_request command for a description of how to execute a network application interactively.

Usage

```
enr <function> {<q_control_args>}
                    {[-arguments | -ag] <function_args>}
```

where:

<function>
: is the name of the network function to be performed. For instance, "ft", or "file_transfer". (This name will be mapped by the Network Information Table to the name of the NASP entry to be called to implement this function.) The functions supported are site dependent. The standard functions supported are documented in a Network User's Manual. Some probable functions are: file_transfer, mail, login (interactive only), l6_attached_processor, cray_attached_processor, and hasp_rje.

<q_control_args> may be any of the following:

-brief, -bf
: suppresses any printed output from the command. The default is to print a line indicating the request_id and how many similar requests are already queued.

-comment <string>, -com <string>
: specifies <string> as a comment to be carried with the queued request. This could be used to convey information to the Multics operator for requests that are deferred. There is no default comment.

-defer_indefinitely, -dfi
: specifies that a queued request is to be deferred until the operator or user determines that it can be run. The default is not to defer the request.

-hold <function> <request_id>
: specifies that this request should not be processed until the request for the specified <function> with the specified <request_id> has been processed. This allows interdependent requests to be entered separately. The default is not to hold the request. This control argument may be repeated to hold the request until a number of other requests have completed.

-long_id, -lgid
: causes the long form of the request id to be printed for a queued request. The default is to print the short form of the request id.

-notify, -nt
: specifies that the user is to be notified when processing of a queued request is started and when it is completed. The default is not to notify the user. This is incompatible with the -notify_start and -notify_end control arguments.

-notify_start, -nts
: specifies that the user is to be notified when the request is first given to the NASP_$execute entry for processing. This is incompatible with the -notify control argument.

-notify_end, -nte
: specifies that the user is to be notified when the NASP_$execute entry indicates that the request is completed. This is incompatible with the -notify control argument.

-proxy USER_ID
: specifies the user for whom this queued request is being entered. Use of this control argument requires additional access to the request queue. The default is the user_id of the process entering the request.

-queue N, -q N
: specifies the priority queue for this request. The default is to enter the request in the default priority queue. The number of queues available and the default queue are specified in the Network Information Table.

-time DT, -tm DT
: specifies the date-time at which this request should

be considered for processing. DT must be in a form acceptable to convert_date_to_binary_. The default is to allow the request to be considered for processing immediately.

<function_args> are described as follows:

These are arbitrary command line arguments (which may look like control arguments) which are passed to, and understood by, the particular network application specified by <function>. The functions supported and the arguments they require are documented in a separate Network User's Manual.

Name: network_request, nr

The network_request command performs a network application interactively in the user's process. See the enter_network_request command for a description of how to queue a request for later execution.

Usage

```
nr <function> {<function_args>}
```

where:

<function>
: is the network application to be performed as described in the enter_network_request command.

<function_args>
: are optional network application specific arguments as described in the enter_network_request command.

Name: cancel_network_request, cnr

The cancel_network_request command allows a previously queued network request to be cancelled and deleted from the request queues. A queued request that is currently running may or may not be cancelled by this command. This ability depends on the particular network application being executed. If more than one request matches the selection criteria given, the matches will be listed and the user will have to pick the one he meant.

Usage

```
cnr <function> {<control_args>}
                {[-arguments | -ag} <function_args>]
```

where:

<function>
: is the name of the network function which was to be performed by the request to be cancelled. See the description in enter_network_request. (Note: the <function> name is mapped to the set of priority queues in which the request resides).

<control_args> may be chosen from the following:

-identifier <request_id>, -id <request_id>
: specifies that <request_id> is the id of the request to be cancelled. This is returned by the enter_network_request command. See the MPM Reference Guide for a description of Request IDs.

-queue N, -q N
: specifies that the request to be cancelled is in priority queue N. If this control argument is omitted, only the default priority queue will be checked.

-all, -a
: specifies that all priority queues should be checked for this request. This control argument is incompatible with the -queue control argument.

-brief, -bf
: suppresses messages telling that a particular request identifier was not found or that requests were cancelled. The default is to print these messages.

-user <user_id>
specifies the name of the submitter of the request to be cancelled, if not specified, the user_id of the current process is assumed. The <user_id> can be of the form: Person_id.Project_id, Person_id, or .Project_id. Both r and d extended access to the queue are required. This control argument is primarily for operators and administrators.

<function_args> are described as follows:

These are arbitrary command line arguments which are passed to, and understood by, the particular network application specified by <function>. They may be used as additional selection criteria for the request to be cancelled as documented in the Network User's Manual.

Name: list_network_request, lnr

The list_network_request command allows the listing of the status of selected queued network requests. The precise information returned about a request is dependent on the particular network application requested. However, any queue related information will always be in a standard format.

Usage

```
lnr <function> {<control_args>}
                    {[-arguments | -ag] <function_args>}
```

where:

<function>
: is the function name of the network requests about which information is to be listed. See the description in enter_network_request.

<control_args> may be chosen from the following:

-identifier <request_id>, -id <request_id>
: specifies that <request_id> is the id of the request to be listed. This is returned by the enter_network_request command. See the MPM Reference Guide for a description of Request IDs.

-queue N, -q N
: specifies that only requests in priority queue N are to be listed. If this control argument is omitted, only requests in the default priority queue will be listed.

-all, -a
: specifies that requests in all priority queues should be listed. This control argument is incompatible with the -queue control argument.

-long, -lg
: causes all of the queue specific information about the request to be listed. This includes time, comment, requests this request is being held for, etc. No request specific information is listed. The default is to only print a line containing the short id, the current state, and the priority of the request.

-request_info, -rqi
: causes any request specific information to be listed. The actual information listed is dependent on the type of request. The default is not to list any request specific information.

-total, -tt
: only lists totals for the requests selected by the other control arguments.

-user <user_id>
: specifies the name of the submitter of the requests to be listed. If not specified, the user_id of the current process is assumed. The <user_id> can be of the form: Person_id.Project_id, Person_id, or .Project_id. Additional access to the queue is required. This control argument is primarily for operators and administrators.

<function_args> are described as follows:

These are arbitrary command line arguments which are passed to, and understood by, the particular network application specified by <function>. Any listing caused by these arguments is application specific and is documented in a Network User's Manual. Any listing information based on these arguments will only be printed if the -request_info control argument is given. However, these arguments will always be used for selecting requests to list.

Name: modify_network_request, mnr

The modify_network_request command allows the queuing parameters of a queued network request to be modified. Application specific parameters of the request may only be changed if the application supports this operation. In particular, it allows requests to be moved between priority queues, and to have their hold statuses changed. If more than one request matches the selection criteria given, the matches will be listed and the user will have to pick the one he meant.

Usage

```
mnr <function> {<q_control_args>}
                    {[-arguments | -ag] <function_args>}
```

where:

<function>
is the name of the function of the request to be modified. See the enter_network_request command for a complete description.

<q_control_args> may be any of the following:

-identifier <request_id>, -id <request_id>
specifies that <request_id> is the id of the request to be modified. This is returned by the enter_network_request command. See the MPM Reference Guide for a description of Request IDs.

-user <user_id>
specifies the name of the submitter of the requests to be modified. If not specified, the user_id of the current process is assumed. The <user_id> can be of the form: Person_id.Project_id, Person_id, or .Project_id. Additional access to the queue is required. This control argument is primarily for operators and administrators.

-comment <string>, -com <string>
specifies that the comment on the request should be changed to <string>.

-defer_indefinitely, -dfi
specifies that the request should be deferred until

the operator or the user determines that it can be run.

-release, -rl
: specifies that a request which specified -defer_indefinitely should be released. This can normally only be done by an operator, administrator, or the user who entered the request.

-hold <function> <request_id>
: specifies that the request should be held until the completion for the specified request. This control argument may be repeated to hold this request for a number of other requests.

-un_hold <function> <request_id>
: specifies that a request being held for the completion of the specified request no longer needs to wait. This control argument may be repeated to un_hold a number of other requests.

-notify, -nt
: specifies that the user wishes to be notified when the processing of this request starts and when it is complete.

-no_notify, -no_nt
: specifies that the user no longer wishes to be notified about the start or completion of the request.

-notify_start, -nts
: specifies that the user wishes to be notified when this request begins execution.

-no_notify_start, -nnts
: specifies that the user does not wish to be notified when this request begins execution.

-notify_end, -nte
: specifies that the user wishes to be notified when this request is completed.

-no_notify_end, -nnte
: specifies that the user does not wish to be notified when this request is completed.

-queue N, -q N
: specifies that the request should be moved to priority queue N.

-time DT, -tm DT
: specifies that the date-time at which this request should be considered for processing should be changed to DT. If DT is 0, then the request may be considered for processing immediately.

<function_args> are described as follows:

These are arbitrary command line arguments which are passed to, and understood by, the particular network application specified by <function>. Any modification caused by these arguments is application specific and is documented in a Network User's Manual.

APPENDIX B

This appendix specifies the interfaces that must be provided by an Network Application Support Procedure (NASP). This includes an entry for processing command or subroutine arguments and building a request structure, and an entry for executing a request, including any code necessary to interface to specific networks.

The subroutine name used in this appendix (NASP_) is for illustration only. The actual name of the NASP to call is found by using the function name to NASP mapping in the Network Information Table. This name is then used with the "network" search list to find the actual subroutine to call. The entries which each NASP must implement have standard names as follows (where NASP_ is replaced by the actual NASP name):

```
NASP_$parser
NASP_$execute
NASP_$cancel
NASP_$modify
NASP_$list
NASP_$info
```

Name: NASP_$parser

This entry takes command line arguments and parses them to build a structure of relevant information that can be used by the NASP_$execute subroutine to carry out the requested function. It is called by the enr or nr command. If it is necessary for the execution of the application, the host name, host attributes, and network name should be kept in the application structure.

Usage

```
dcl NASP_$parser entry (ptr, ptr, char (*), ptr, fixed bin,
                        bit (1), ptr, ptr, fixed bin (24),
                        char (*) varying, fixed bin (35));

call NASP_$parser (in_iocbp, out_iocbp, caller_name, arg_list_ptr,
                   first_arg, queued_flag, area_ptr,
                   structure_ptr, structure_len,
                   error_msg, code);
```

where:

in_iocbp (Input)
is a pointer to an I/O switch to be used if input from the user is necessary. It will be open for stream_input.

out_iocbp (Input)
is a pointer to an I/O switch to be used if output to the user is necessary. It will be open for stream_output.

caller_name (Input)
is the name of the caller which should be used in reporting errors or querying the user.

arg_list_ptr (Input)
is a pointer to the user supplied command arguments for this function. The cu_$xxx_rel entry points should be used to access the arguments.

first_arg (Input)
is the number of the first argument after the -arguments control argument on the command line (i.e. the first function specific argument). If there are no arguments after the -arguments control argument,

or there is no -arguments control argument, then this will be zero.

queued_flag (Input)
is a flag indicating whether the user command called for queuing this request or for execution in the user's process.

"1"b -- this request will be queued.
"0"b -- this request will be executed interactively.

area_ptr (Input)
is a pointer to an area in which to allocate the structure to be returned. If this pointer is null, then the system free area should be used.

structure_ptr (Output)
is a pointer to the structure which has been built to hold the request information. This structure is not interpreted by the caller, but is passed as is to other NASP entries.

structure_len (Output)
is the length of the returned structure in bits.

error_msg (Output)
is an error message text that more fully explains an error. It will only be used by the caller if code is non-zero. The minimum length of the string on input will be 256 characters.

code (Output)
is a standard system status code. The specific codes returned are NASP dependent.

Name: NASP_$execute

This subroutine takes an application specific request structure, built by NASP_$parser, and performs the requested action. This subroutine contains any network or host specific knowledge needed to perform the function. It is called by the nr command, or by a daemon process when a queued request is to be executed.

Usage

```
dcl NASP_$execute entry (ptr, ptr, char (*), bit (1), ptr,
                          fixed bin (24), bit (1), bit (1),
                          char (*) varying, fixed bin (35));

call NASP_$execute (in_iocbp, out_iocbp, caller_name, queued_flag,
                    structure_ptr, structure_len, complete,
                    unhold, error_msg, code);
```

where:

in_iocbp (Input)
is a pointer to an I/O switch to be used if input from the user is necessary. It will be open for stream_input.

out_iocbp (Input)
is a pointer to an I/O switch to be used if output to the user is necessary. It will be open for stream_output.

caller_name (Input)
is the name of the caller which should be used in reporting errors or querying the user.

queued_flag (Input)
is a flag which indicates whether this request came from a queue or directly from the user's process.

"1"b -- this request came from a queue.
"0"b -- this request is being done interactively.

structure_ptr (Input/Output)
on input, is a pointer to the structure which has been built to hold the request information. This structure was presumably built by the NASP_$parser subroutine. On output, the structure holds the

(possibly updated) request information which will be put back in the request queue.

structure_len (Input/Output)
is the length of the structure in bits. This must be the same on input and output.

complete (Output)
indicates whether all processing for this request is completed or not completed. If the request is completed, it will be deleted from the queue. If the request is not complete, it will remain in the queue for later processing after being updated (by the caller) using the output values of structure_ptr and structure_len.

"1"b -- the request is complete.
"0"b -- the request is not complete.

unhold (Output)
indicates whether any requests which are being held for the completion of this request should be updated to reflect the completion of this request.

"1"b -- update held requests.
"0"b -- do not update held requests.

error_msg (Output)
is an error message text that more fully explains an error. It will only be used by the caller if code is non-zero. The minimum length of the string on input will be 256 characters.

code (Output)
is a standard system status code. The specific codes returned are NASP dependent.

Name: NASP_$cancel

This subroutine is called by the cnr command or an operator command in a daemon process when the user desires to cancel a request whose state is "ready" or "not completed". If a zero error code is returned, indicating sucessful cancellation, the queue management software will then delete the request from the queue. If a non-zero error code is returned, indicating that the request was not cancelled, the error code will be returned to the user, and the request will be left (updated) in the queue. A request in the "deferred" state will simply be deleted from the queue without calling any NASP entry.

Usage

```
dcl NASP_$cancel entry (ptr, fixed bin (24), char (*) varying,
                        fixed bin (35));

call NASP_$cancel entry (structure_ptr, structure_len, error_msg,
                         code);
```

where:

structure_ptr (Input/Output)
on input, is a pointer to the structure which has been built to hold the request information. This structure was presumably built by the NASP_$parser subroutine. On output, the struture holds the (possibly updated) request information which will be put back in the request queue if the cancellation failed.

structure_len (Input/Output)
is the length of the structure in bits. This must be the same on input and output.

error_msg (Output)
is an error message text that more fully explains an error. It will only be used by the caller if code is non-zero. The minimum length of the string on input will be 256 characters.

code (Output)
is a standard system status code. The specific codes returned are NASP dependent.

Name: NASP_$modify

This subroutine takes command line arguments and an existing request structure and makes modifications to the request based on the arguments. It is called by the mnr command which will then update the request in the queue by using the output from this entry.

Usage

```
dcl NASP_$modify entry (ptr, ptr, char (*), ptr, fixed bin,
                        ptr, fixed bin (24), char (*) varying,
                        fixed bin (35));

call NASP_$modify (in_iocbp, out_iocbp, caller_name, arg_list_ptr,
                   first_arg, structure_ptr, structure_len,
                   error_msg, code);
```

where:

in_iocbp (Input)
: is a pointer to an I/O switch to be used if input from the user is necessary. It will be open for stream_input.

out_iocbp (Input)
: is a pointer to an I/O switch to be used if output to the user is necessary. It will be open for stream_output.

caller_name (Input)
: is the name of the caller which should be used in reporting errors or querying the user.

arg_list_ptr (Input)
: is a pointer to the user supplied command arguments for this function. The cu_$xxx_rel entry points should be used to access the arguments.

first_arg (Input)
: is the number of the first argument after the -arguments control argument on the command line (i.e. the first function specific argument). If there are no arguments after the -arguments control argument, or there is no -arguments control argument, then this will be zero.

structure_ptr (Input/Output)
on input, is a pointer to the structure which has been built to hold the request information. This structure was presumably built by the NASP_$parser subroutine. On output, the struture holds the modified request information which will be put back in the request queue.

structure_len (Input/Output)
is the length of the structure in bits. This must be the same on input and output.

error_msg (Output)
is an error message text that more fully explains an error. It will only be used by the caller if code is non-zero. The minimum length of the string on input will be 256 characters.

code (Output)
is a standard system status code. The specific codes returned are NASP dependent.

Name: NASP_$list

This subroutine takes command line arguments and an existing request structure and lists information about the request based on the arguments. It is called by the lnr command or by an operator command in a daemon process. It will be called once for each request which matches the user specified selection criteria. If the user has also specified application specific selection criteria, this entry interprets them and indicates whether this request should be listed or not.

Usage

```
dcl NASP_$list entry (ptr, fixed bin, ptr, fixed bin (24),
                      bit (1), char (*) varying,
                      char (*) varying, fixed bin (35));

call NASP_$list (arg_list_ptr, first_arg, structure_ptr,
                 structure_len, list, output_lines,
                 error_msg, code);
```

where:

arg_list_ptr (Input)
: is a pointer to the user supplied command arguments for this function. The cu_$xxx_rel entry points should be used to access the arguments.

first_arg (Input)
: is the number of the first argument after the -arguments control argument on the command line (i.e. the first function specific argument). If there are no arguments after the -arguments control argument, or there is no -arguments control argument, then this will be zero.

structure_ptr (Input)
: is a pointer to the structure which holds the request to be listed.

structure_len (Input)
: is the length of the structure in bits.

list (Output)
: indicates whether or not this request should be listed (based on the application specific arguments).

"1"b -- list this request.
"0"b -- do not list this request.

output_lines (Output)
is a character string containing the information to be listed. It may contain newline characters and will be output to the user, possibly indented, with an ioa_ "^a" control string.

error_msg (Output)
is an error message text that more fully explains an error. It will only be used by the caller if code is non-zero. The minimum length of the string on input will be 256 characters.

code (Output)
is a standard system status code. The specific codes returned are NASP dependent.

Name: NASP_$info

This entry takes an application specific request structure and returns information about the request. Currently, the only information returned is the host name and the network name for the request.

Usage

```
dcl NASP_$info entry (ptr, fixed bin (24), ptr, char (*) varying,
                      fixed bin (35));

call NASP_$list (structure_ptr, structure_len, nasp_info_ptr,
                 error_msg, code);
```

where:

structure_ptr (Input)
: is a pointer to the structure which holds the request for which information is desired.

structure_len (Input)
: is the length of the structure in bits.

nasp_info_ptr (Input/Output)
: is a pointer to a structure in which the information is returned. The structure is described in Notes below. The version field must be set on input, all other fields will be set on output.

error_msg (Output)
: is an error message text that more fully explains an error. It will only be used by the caller if code is non-zero. The minimum length of the string on input will be 256 characters.

code (Output)
: is a standard system status code. The specific codes returned are NASP dependent.

Notes

The following declaration describes the nasp_info structure (declared in nasp_info.incl.pl1):

```
dcl 1 nasp_info aligned based,
     2 version fixed bin (35),
     2 host_name char (32) unaligned,
     2 net_name char (32) unaligned;
```

where:

version
: is the version number of this structure. It should be set to nasp_info_v1 on input.

host_name
: is the host name associated with this request. If it is all blanks, then there is no host name for this request.

net_name
: is the network name associated with this request. If it is all blanks, then there is no network name for this request.

APPENDIX C

This appendix specifies the interface being proposed for entering and retrieving information from the Multics Network Information Table. The NIT will be implemented using a MRDS database. The following entries are defined to allow easy access to the database. Notice that the MRDS accessing subroutines may be used directly by the user to retrieve information in ways not provided by these entries. The ACLs on the nit_ entries will probably give "e" access to everyone, the ACLs for the nit_admin_ entries will probably restrict "e" access to administrators.

```
nit_$get_function_address
nit_$get_host_address
nit_$get_host_attributes
nit_$get_nasp_name
nit_$get_net_attributes
nit_$get_q_pathname
```

```
nit_admin_$set_function_address
nit_admin_$set_host_address
nit_admin_$set_host_attribute
nit_admin_$set_nasp_name
nit_admin_$set_net_attribute
nit_admin_$set_q_pathname
```

The source for generating the MRDS database is given below:

```
domain:

/* The primary names and any secondary names for various       */
/* entities.                                                    */

          host_name           char (32) unaligned,
          net_name            char (32) unaligned,
          function_name       char (32) unaligned,

/* Id's so that each entity has a unique name.                  */
/* These will be generated using unique_bits_.                  */

          host_id             bit (70),
          net_id              bit (70),
          function_id         bit (70),

/* Attributes and host address values.                          */

          host_attribute      char (32) unaligned,
          net_attribute       char (32) unaligned,
          net_address         char (200) varying aligned,

/* This is mostly to give the mailbox name of a function in DSA.*/

          function_address    char (32) unaligned,

/* Information to find the queues.  There is one queue per      */
/* function and priority.  A priority of zero will give the     */
/* default queue.                                               */

          queue_pathname      char (168) varying aligned,
          queue_priority      fixed bin (17) unaligned,

/* Name of the NASP function.                                   */

          nasp_name           char (32) unaligned;

relation:

/* Name to identifier mapping relations                         */

          host_names (host_name* host_id),
          net_names (net_name* net_id),
          function_names (function_name* function_id),

/* Information retrieval relations                              */

          net_attributes (net_id* net_attribute*),
          host_attributes (host_id* host_attribute*),
          host_address (host_id* net_id* net_address),

          sub_address (host_id* net_id* function_id*
                        function_address),

          queue (function_id* queue_priority*
                  queue_pathname),

          nasp_names (function_id* nasp_name);

index:
          host_names (host_id),
          net_names (net_id),
          function_names (function_id),

          net_attributes (net_attribute),
          host_attributes (host_attribute),
          host_address (net_id net_address),
          sub_address (net_id function_id),
          nasp_names (nasp_name);
```

Name: nit_$get_function_address

This entry takes a function name, host name, and network name, and returns any function specific addressing information needed to access that function.

Usage

```
dcl nit_$get_function_address entry (char (*), char (*), char (*),
                                      char (*), fixed bin (35));

call nit_$get_function_address (function, host, net, address, code);
```

where:

function (Input)
: is the function name of the function whose address is desired.

host (Input)
: is the name of the host on which the function is to be performed.

net (Input)
: is the name of the network over which the request for function execution will be sent.

address (Output)
: is the addressing information needed to address the particular function on the specified host over the specified network.

code (Output)
: is a standard system status code.

Name: nit_$get_host_address

This entry takes a host name and network name and returns the address of that host on that network.

Usage

```
dcl nit_$get_host_address entry (char (*), char (*), char (*),
                                  fixed bin (35));

call nit_$get_host_address (host, net, address, code);
```

where:

host (Input)
is a character string name of a host.

net (Input)
is the name of the network for which the host address is desired.

address (Output)
is the address of the host on the network.

code (Output)
is a standard system status code.

Name: nit_$get_host_attributes

This entry takes a host name and returns the attributes of that host. The attributes are returned in an attribute string which can be manipulated by the mode_string_ subroutine entries. This will require this entry to make multiple database accesses to retrieve all of the host attributes.

Usage

```
dcl nit_$get_host_attributes entry (char (*), char (*),
                                    fixed bin (35));

call nit_$get_host_attributes (host, attributes, code);
```

where:

host (Input)
is a character string name of a host.

attributes (Output)
are the attributes of the specified host in a character string suitable for manipulation by the mode_string_ subroutine entries. If the output character string is not long enough to contain all of the attributes, then those that fit will be returned and an error code will also be returned.

code (Output)
is a standard system status code.

Name: nit_$get_nasp_name

This entry takes a user specified function name and returns the name of the NASP entry which implements that function. This name should then be used with the "network" search list to find the subroutine to call. This subroutine must have the standard entries as defined in the Network User's Guide. This mapping is site specifiable (see the description for Network Information Table administration in the MAM).

Usage

```
dcl nit_$get_nasp_name entry (char (*), char (*), fixed bin (35));

call nit_$get_nasp_name (function, nasp_name, code);
```

where:

function	(Input) is a character string name of a function to be performed.
nasp_name	(Output) is the name of the NASP which implements the specified function.
code	(Output) is a standard system status code.

Name: nit_$get_net_attributes

This entry takes a network name and returns the attributes of that network. The attributes are returned in an attribute string which can be manipulated by the mode_string_ subroutine entries. This will require this entry to make multiple database accesses to retrieve all of the network attributes.

Usage

```
dcl nit_$get_net_attributes entry (char (*), char (*),
                                   fixed bin (35));

call nit_$get_net_attributes (net, attributes, code);
```

where:

net	(Input) is a character string name of a network.
attributes	(Output) are the attributes of the specified network in a character string suitable for manipulation by the mode_string_ subroutine entries. If the output character string is not long enough to contain all of the attributes, then those that fit will be returned and an error code will also be returned.
code	(Output) is a standard system status code.

Name: nit_$get_q_pathname

This entry takes a user specified function name and priority, and returns the pathname of the request queue which holds these types of requests. This mapping is site specifiable (see the description for Network Information Table administration in the MAM).

Usage

```
dcl nit_$get_q_pathname entry (char (*), fixed bin, char (168),
                                        char (32), fixed bin(35));

call nit_$get_q_pathname (function, priority, q_dirname,
                                   q_entryname, code);
```

where:

function (Input)
: is a character string name of a function to be performed.

priority (Input)
: is the priority number of the queue desired. If this is zero then the pathname of the default priority queue will be returned.

q_dirname (Output)
: is the directory name portion of the pathname of the queue in which requests for the specified function and priority reside.

q_entryname (Output)
: is the entryname portion of the pathname of the queue in which requests for the specified function and priority reside.

code (Output)
: is a standard system status code.

Name: nit_admin_$set_function_address

This entry takes a function name, host name, and network name, and sets the function specific addressing information needed to access that function.

Usage

```
dcl nit_admin_$set_function_address entry (char (*), char (*), char (*),
                              char (*), fixed bin (35));

call nit_admin_$set_function_address (function, host, net, address,
                                   code);
```

where:

function (Input)
is the function name of the function whose address is to be set.

host (Input)
is the name of the host on which the function is to be performed.

net (Input)
is the name of the network over which the request for function execution will be sent.

address (Input)
is the addressing information needed to address the particular function on the specified host over the specified network.

code (Output)
is a standard system status code.

Name: nit_admin_$set_host_address

This entry takes a host name and network name and sets the address of that host on that network.

Usage

```
dcl nit_admin_$set_host_address entry (char (*), char (*), char (*),
                                       fixed bin (35));

call nit_admin_$set_host_address (host, net, address, code);
```

where:

host (Input)
is a character string name of a host whose address is to be set.

net (Input)
is the name of the network for which the host address is to be set.

address (Input)
is the address to be set of the host on the network.

code (Output)
is a standard system status code.

Name: nit_admin_$set_host_attribute

This entry takes a host name and sets one attribute for that host. The attribute should have a syntax like the syntax for a mode in a standard system mode string. Multiple calls to this entry are necessary to set multiple attributes for a host.

Usage

```
dcl nit_admin_$set_host_attribute entry (char (*), char (*),
                                          fixed bin (35));

call nit_admin_$set_host_attribute (host, attribute, code);
```

where:

host	(Input) is a character string name of a host.
attribute	(Input) is an attribute for the host (with a syntax like that of a mode in a standard system mode string).
code	(Output) is a standard system status code.

Name: nit_admin_$set_nasp_name

This entry takes a user specified function name and sets the name of the NASP entry which implements that function. This name will be used with the "network" search list to find the subroutine implementing this function.

Usage

```
dcl nit_admin_$set_nasp_name entry (char (*), char (*),
                                    fixed bin (35));

call nit_admin_$set_nasp_name (function, nasp_name, code);
```

where:

function (Input)
is a character string name of a function to be performed.

nasp_name (Input)
is the name of the NASP which implements the specified function.

code (Output)
is a standard system status code.

Name: nit_admin_$set_net_attribute

This entry takes a network name and sets one attribute for that network. The attribute should have a syntax like the syntax for a mode in a standard system mode string. Multiple calls to this entry are necessary to set multiple attributes for a network.

Usage

```
dcl nit_admin_$set_net_attribute entry (char (*), char (*),
                                           fixed bin (35));

call nit_admin_$set_net_attribute (net, attribute, code);
```

where:

net (Input)
is a character string name of a network.

attribute (Input)
is an attribute for the network (with a syntax like that of a mode in a standard system mode string).

code (Output)
is a standard system status code.

Name: nit_admin_$set_q_pathname

This entry takes a user specified function name and priority, and sets the pathname of the request queue which holds these types of requests.

Usage

```
dcl nit_admin_$set_q_pathname entry (char (*), fixed bin,
                                     char (*), fixed bin(35));

call nit_admin_$set_q_pathname (function, priority,
                                q_pathname, code);
```

where:

function (Input)
is a character string name of a function to be performed.

priority (Input)
is the priority number of the queue desired. If this is zero then this will be the pathname of the default priority queue.

q_pathname (Input)
is the pathname of the request queue in which requests for the specified function and priority should be put.

code (Output)
is a standard system status code.

APPENDIX D

This appendix specifies the interface being proposed for entering and retrieving requests from request queues. The implementation details of the request queues are also specified. The following entries are described:

nrq_manager_$put
nrq_manager_$get
nrq_manager_$update
nrq_manager_$delete

Queue Implementation Details

The queues are implemented using the standard message segment primitives. However, a new primitive will be necessary to allow a request to be read and locked in a single "atomic" operation. This will allow multiple processes to use a single queue for requests without having to use a "coordinator" process.

Name: nrq_manager_$put

This entry takes a request, verifies that the request header is in the proper format and puts it in the proper network request queue.

Usage

```
dcl nrq_manager_$put entry (ptr, fixed bin(35));

call nrq_manager_$put (request_ptr, code);
```

where:

request_ptr (Input/Output)
is a pointer to the request to be queued. The first part of the request is expected to be a request header in a standard format (see Notes below). The time_entered, request_id, q_dirname, and q_entryname fields will be filled in on return. If the priority field is 0, then the default priority queue will be used, and its number will be filled in. All other fields will be left alone. Note: the state of the request must be "deferred" or "ready" to be sucessfully put into the queue.

code (Output)
is a standard system status code.

Notes

The following pl1 declaration (in network_request.incl.pl1) specifies the structure of a network request. Included in a request is a standard request header which contains information which is interpreted by queue management routines, and application specific request information which is interpreted by NASPs.

```
dcl 1 network_request          aligned based,
     2 standard_header         aligned,
      3 version                fixed bin (35),
      3 time_entered           fixed bin (71),
      3 request_id             fixed bin (71),
      3 entered_by             char (32) unaligned,
      3 entered_for            char (32) unaligned,
      3 function_id            bit (70) aligned,
      3 priority               fixed bin (17) unaligned,
      3 q_dirname              char (168) unaligned,
      3 q_entryname            char (32) unaligned,
      3 state                  fixed bin (18) unsigned unaligned,
      3 comment                char (128) varying unaligned,
      3 defer_until            fixed bin (71),
      3 grace_time_expires     fixed bin (71),
      3 hold_array_len         fixed bin (17) aligned,
      3 hold_array (ha_len refer (network_request.hold_array_len)),
       4 hold_function_id      bit (70) aligned,
       4 hold_request_id       fixed bin (71) aligned,
       4 flags                 aligned,
        5 request_complete     bit (1) unaligned,
        5 pad                  bit (35) unaligned,
      3 flags                  aligned,
       4 defer_indefinitely    bit (1) unaligned,
       4 notify_start          bit (1) unaligned,
       4 notify_end            bit (1) unaligned,
       4 pad                   bit (33) unaligned,
      3 application_info_len   fixed bin (24),
     2 application_info        bit (nra_len refer
                               network_request.application_info_len);
```

where:

version
: is the version of this structure, it should be set to network_request_v1.

time_entered
: is the time the request was entered in the queue.

request_id
: is the identifier for this request, as returned by the queue management software.

entered_by
: is the user_id of the user who entered the request.

entered_for
: is the user_id of the user for whom the request was entered. This will be the same as the entered_by field for non-priviledged users.

function
: is the unique identifier of the function performed by this request.

priority
: is the priority number of the queue in which this request resides.

q_dirname
: is the directory part of the pathname of the queue in which this request resides.

q_entryname
: is the entryname part of the pathname of the queue in which this request resides.

state
: is the current state of this request, encoded as follows:

  1 - unprocessed (means that the request has not been looked at since it was entered in the queue.)
  2 - deferred (means the request is waiting for a certain time, or for operator or user action.)
  3 - ready (means the request is ready to be run.)
  4 - running (means the request is currently being executed. It will still be locked in the queue in this state.)
  5 - not complete (means the request has been partially executed, but is not yet finished.)
  6 - aborted (means that the system crashed while the request was being executed, or while it was locked, and that it may be in an inconsistent state.)

  (Note: a request may have an application specific state which is kept as part of the application information.)

comment
: is the comment associated with this request.

defer_until
: is the time until which this request was deferred by the user. It can be executed after this time. This may be 0 if no time was specified by the user.

grace_time_expires
: is the time after which the request may be deleted from the queue. It is only valid for a request in the "finished" state.

hold_array_len
: is the number of entries in the hold array.

hold_function_id
: is the unique identifier of the function which will be performed by the request for which this request is being held.

hold_request_id
: is the request_id of the request for which this request is being held.

request_complete
: is a flag which is set to "1"b if the request described by this array entry is complete. Otherwise, it is "0"b.

defer_indefinitely
: indicates, if "1"b, that this request is deferred by the user until the operator releases it.

notify_start
: indicates, if "1"b, that the user wishes to be notified when the request starts execution (i.e. enters the "running" state).

notify_end
: indicates, if "1"b, that the user wishes to be notified when the request ends execution (i.e. enters the "finished" state).

application_info_len
: is the length of the application specific information in bits.

application_info
: is a bit string which holds the application specific information. This information is not looked at by the queue management software, but is passed as is to the various NASP entries.

Name: nrq_manager_$get

This entry gets a copy of a request from a queue which matches the selection criteria. The request is left locked (by the calling process) in the queue. It is normally unlocked by the nrq_manager_$update entry, however, all the entries will handle invalid locks. (Note: the standard network request queue header contains the pathname of the queue from which the request was taken).

Usage

```
dcl nrq_manager_$get entry (char (*), fixed bin (17), bit (1),
                        fixed bin (71), char (*),
                        ptr, ptr, fixed bin (35));

call nrq_manager_$get (function, priority, from_beginning,
                    request_id, user, area_ptr, request_ptr,
                    code);
```

where:

function (Input)
is a character string name of a function to be performed.

priority (Input)
is the priority number of the queue from which to get the request.

from_beginning (Input)
is a flag indicating whether the specified queue should be scanned from the beginning ro from the last request seen by this process.

"1"b -- start the scan at the beginning of the queue.
"0"b -- start the scan from the last request in the queue seen by this process.

request_id (Input)
is the id of the desired request. The id may be 0 if no request_id is to be specified.

user (Input)
is the user_id of the user for which the desired request was entered. This may be of the form: Person_id.Project_id, Person_id, or .Project_id. If

this argument is all blanks, then the user_id of the current process is assumed. This will be checked against the "entered_for" field in the request.

area_ptr (Input)
is a pointer to an area in which space for the returned request should be allocated. If this pointer is null then the system free area will be used.

request_ptr (Output)
is a pointer to the returned request. See the nrq_manager_$put entry for a description of the request structure.

code (Output)
is a standard system status code.

Name: nrq_manager_$update

This entry takes a request, verifies that the request header is consistent with the request in the queue, and then replaces the request in the queue with the given request, unlocking it in the process. The standard request header contains the information necessary to locate the request to update.

Usage

```
dcl nrq_manager_$update entry (ptr, fixed bin (35));

call nrq_manager_$update (request_ptr, code);
```

where:

request_ptr (Input)
is a pointer to the updated copy of the request. See the nrq_manager_$put entry for a description of the request structure.

code (Output)
is a standard system status code.

Name: nrq_manager_$delete

This entry takes a request, verifies that the request header is consistent with the queued request, and deletes the request from the queue. The standard request header contains the information necessary to locate the request to update.

Usage

```
dcl nrq_manager_$delete entry (ptr, fixed bin (35));

call nrq_manager_$delete (request_ptr, code);
```

where:

request_ptr (Input)
is a pointer to the copy of the request to be deleted. See the nrq_manager_$put entry for a description of the request structure.

code (Output)
is a standard system status code.

APPENDIX E

This appendix specifies the commands which may be used to control the request selection criteria and other aspects of the daemon operation. The command environment will be implemented using the subsystem utilities. The start_up.ec will just set things up so that the subsystem can operate as a closed subsystem.

Name: handle_requests_for, hrf

This command allows an operator to add to the types of requests that this daemon should handle.

Usage

```
hrf <function> <host> <network>
```

where:

<function>
: is the name of an application function that this daemon should handle. It may be specified as "*", in which case any function may be handled.

<host>
: is the name of a host which this daemon should handle. It may be specified as "*", in which case any host may be handled.

<network>
: is the name of a network which this daemon should handle. It may be specified as "*", in which case any network may be handled.

Name: dont_handle, dh

This command specifies a function, host, and network combination that will not be handled by this daemon.

Usage

```
dh <function> <host> <network>
```

where:

<function>
: is the name of a function which should not be handled. It may not be "*" since this would make any processing impossible.

<host>
: is the name of a host which should not be handled. It may not be "*" since this would make any processing impossible.

<network>
: is the name of a network which should not be handled. It may not be "*" since this would make any processing impossible.

Name: list, ls

This command lists all the function, host, and network combinations that this daemon currently handles. It also can be used to list requests that this daemon can handle.

Usage

```
ls {<control_args>}
```

where:

<control_args> may be chosen from the following: (if no control_args are specified, then all the function, host, and network combination handled by this daemon are listed.)

-user USER_ID
: lists the request for the specified user that exist in the queues handled by this daemon.

-defer_indefinitely, -dfi
: only lists request which were entered with the -defer_indefinitely control argument and therefore, require operator action to start.

Name: go

This command causes the daemon to start processing requests according to the current function, host, and network combinations.

Usage

```
go
```

Name: release

This command releases a request which is being held because the user entered it with the -defer_indefinitely control argument.

Usage

```
release request_id user_id
```

where:

request_id
: is the request id of the request to be released. It can be found by using the list command.

user_id
: is the user_id of the user who entered the request to be released. This is used as a check.

Name: cancel

This command cancels a request and deletes it from the request queue. A request whose state is "running" may be left in an inconsistent state depending on the particular application.

Usage

```
cancel request_id user_id
```

where:

request_id
: is the request id of the request to be cancelled. It can be found by using the list command.

user_id
: is the user_id of the user who entered the request to be cancelled. This is used as a check.

APPENDIX F

This appendix gives MAM style documentation describing the administrative operations which can be performed to control the operation of the networking facility described in this MTB.

APPENDIX G

This appendix describes the arguments accepted by the file_transfer (ft) function to allow file transfers over the ARPANET, a direct or dial_up connection, and a DSA network.

Name: file_transfer, ft

This network function allows user's to transfer files over any network connection to or from a host on that network. This function must be used in conjunction with the network_request commands: enr, nr, lnr, cnr, and mnr. These commands are described in the MPM commands manual.

Usage

```
enr ft <q_control_args> {[-arguments | -ag] <function_args>}

nr ft <function_args>
```

where:

<q_control_args>
are queuing control arguments described for the enr command.

<function_args> have the following structure:

```
     <source_file> <destination_file> {<control_args>}
```

where:

<source_file> ::= {-name | -nm} <file_name> {-at <host>}
specifies the source file to be used for the transfer. <file_name> must be preceded by -name or -nm if it begins with a "-". It must be enclosed in quotes if it contains spaces or special characters. It must be followed by "-at <host>" if the file does not reside on the local host. The <file_name> is specified in a syntax acceptable to the host on which the file resides. If the <host> is a Multics system, then the <file_name> may be a "star" name, and all of the files which match the "star" name will be transferred.

<destination_file> ::= {-name | -nm} <file_name> {-at <host}
specifies the destination file to be used for the transfer. It has the same syntax and restrictions as <source_file>. If the <host> is a Multics system, then the <file_name> may be an "equal" name, and the actual file names will be generated using the equals convention.

<control_args> may be chosen from the following:

-network NAME, -net NAME
: specifies the particular network to use for the transfer. If no network is specified then any appropriate and available network will be used.

-user STR
: STR specifies the user on whose behalf the file transfer is to be done. This may be used by the remote host for authentication of the file transfer. The default is the Multics user_id of the user who submitted the request.

-password STR, -pw STR
: STR is a password that may be used by the remote host to authenticate the file transfer. There is no default. If the remote host requires a password, and none is given, then the user will be prompted for one with a mask.

-compression, -cmpr
: specifies that the file should be compressed for the transfer to allow more efficient use of the line. This is the default. See "Notes" below.

-no_compression, -no_cmpr
: specifies that the file should not be compressed for the transfer.

-checkpoint_interval N, -ci N
: specifies that checkpoint marks should be used every N records during the transfer. The default is not to use checkpointing. See "Notes" below.

-delete, -dl
: mark the source file for deletion after the transfer has been completed. The file will actually be deleted only if the transfer was successful and a grace period has elapsed. The default is not to delete the source file.

-data_type [ascii ¦ binary ¦ ebcdic]
: specifies the data type to be used for the transfer. The default is "binary".

-append
: specifies that the source file should be appended to

the destination file, rather that replacing it. See "Notes" below.

-force, -fc
: specifies that the destination file is to be written even if it already exists. The default is not to overwrite an existing file. If the destination host is a Multics system, then standard namedup handling will be used as the default.

Notes

All networks may not support all the control arguments listed above. If the user specifies the network using -network or -net, then any control arguments not supported will cause the rejection of the request. If the network is not specified, then any control arguments not supported will be ignored.

In the first implementation, there is a restriction that either the source file or the destination file must be on the local host (i.e. both must not use the -at argument). Thus, third-party transfers are not allowed.