JOHNSON GRANT
IN-61-CR
312534
P.98

DATA MANAGEMENT SYSTEM (DMS) TESTBED USER'S MANUAL DEVELOPMENT

(NASA-CR-187391) DATA MANAGEMENT SYSTEM (DMS) TESTBED USER'S MANUAL DEVELOPMENT, VOLUMES 1 AND 2 (Softech) 98 p CSCL 09B

N91-13093

Unclas

G3/61 0312534

J. G. McBride

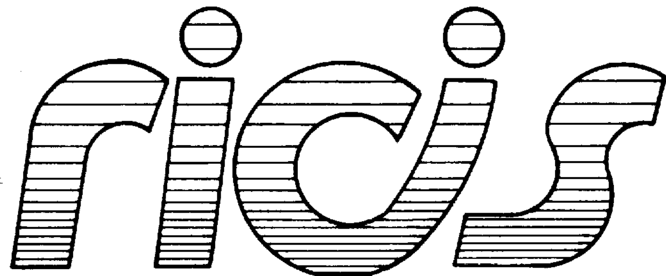
SofTech, Inc.

N. Cohen

SofTech, Inc.

October 31, 1986

Cooperative Agreement NCC 9-16
Research Activity No. *SE.2*



Research Institute for Computing and Information Systems
University of Houston - Clear Lake

T·E·C·H·N·I·C·A·L R·E·P·O·R·T

The RICIS Concept

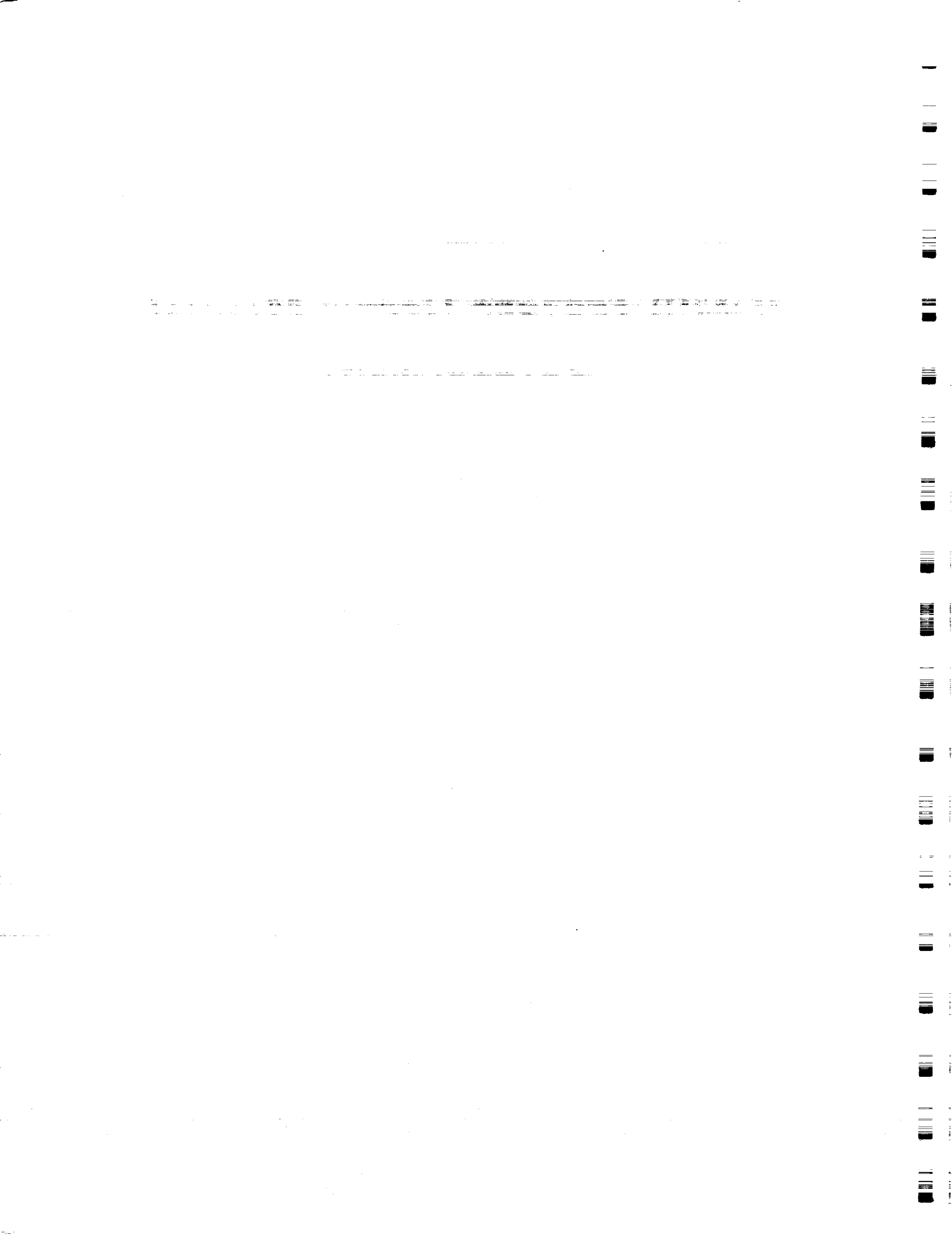
The University of Houston-Clear Lake established the Research Institute for Computing and Information systems in 1986 to encourage NASA Johnson Space Center and local industry to actively support research in the computing and information sciences. As part of this endeavor, UH-Clear Lake proposed a partnership with JSC to jointly define and manage an integrated program of research in advanced data processing technology needed for JSC's main missions, including administrative, engineering and science responsibilities. JSC agreed and entered into a three-year cooperative agreement with UH-Clear Lake beginning in May, 1986, to jointly plan and execute such research through RICIS. Additionally, under Cooperative Agreement NCC 9-16, computing and educational facilities are shared by the two institutions to conduct the research.

The mission of RICIS is to conduct, coordinate and disseminate research on computing and information systems among researchers, sponsors and users from UH-Clear Lake, NASA/JSC, and other research organizations. Within UH-Clear Lake, the mission is being implemented through interdisciplinary involvement of faculty and students from each of the four schools: Business, Education, Human Sciences and Humanities, and Natural and Applied Sciences.

Other research organizations are involved via the "gateway" concept. UH-Clear Lake establishes relationships with other universities and research organizations, having common research interests, to provide additional sources of expertise to conduct needed research.

A major role of RICIS is to find the best match of sponsors, researchers and research objectives to advance knowledge in the computing and information sciences. Working jointly with NASA/JSC, RICIS advises on research needs, recommends principals for conducting the research, provides technical and administrative support to coordinate the research, and integrates technical results into the cooperative goals of UH-Clear Lake and NASA/JSC.

DATA MANAGEMENT SYSTEM (DMS) TESTBED USER'S MANUAL DEVELOPMENT

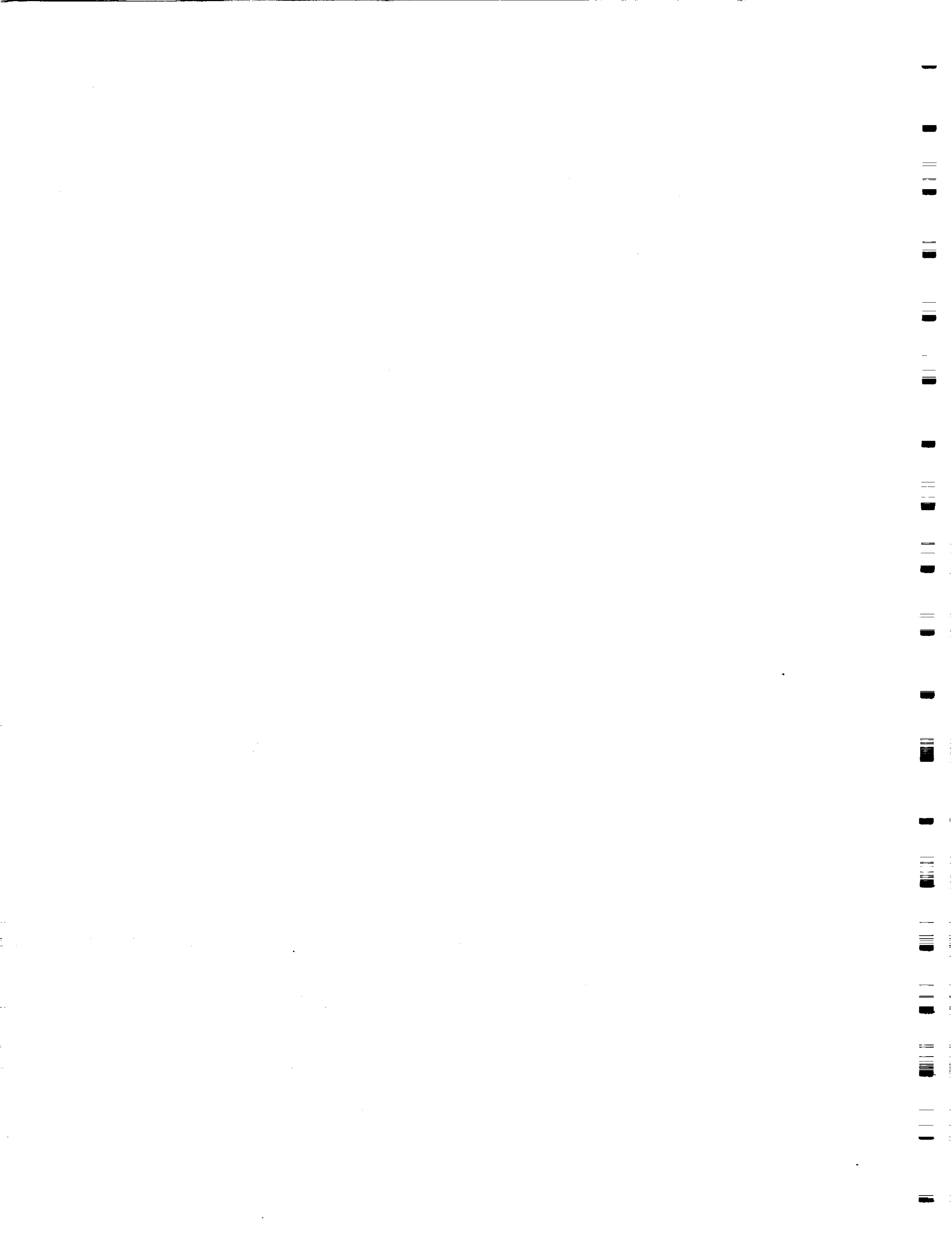


Preface

This research was conducted under the auspices of the Research Institute for Computing and Information Systems by SofTech, Inc. The project was under the overall technical direction of Dr. Charles McKay, Director, Software Engineering Research Center at the University of Houston-Clear Lake. Primary research for this project was done by John McBride, Manager of the SofTech Houston Operations and Norman Cohen, SofTech Systems Consultant.

Funding was provided by the Avionics System Division, Engineering Directorate, NASA Johnson Space Center through Cooperative Agreement NCC 9-16 between NASA/JSC and UH-Clear Lake. The NASA Technical Monitor for this activity was Gary K. Raines, Head, Data Processing Section.

The views and conclusions contained in this report are those of the authors and should not be interpreted as representative of the official policies, either express or implied, of NASA or the United States Government.



SOFTECH

DATA MANAGEMENT SYSTEM (DMS)
TESTBED USER'S MANUAL
DEVELOPMENT

VOLUMES I & II

WO-092

31 OCTOBER 1986

Prepared for

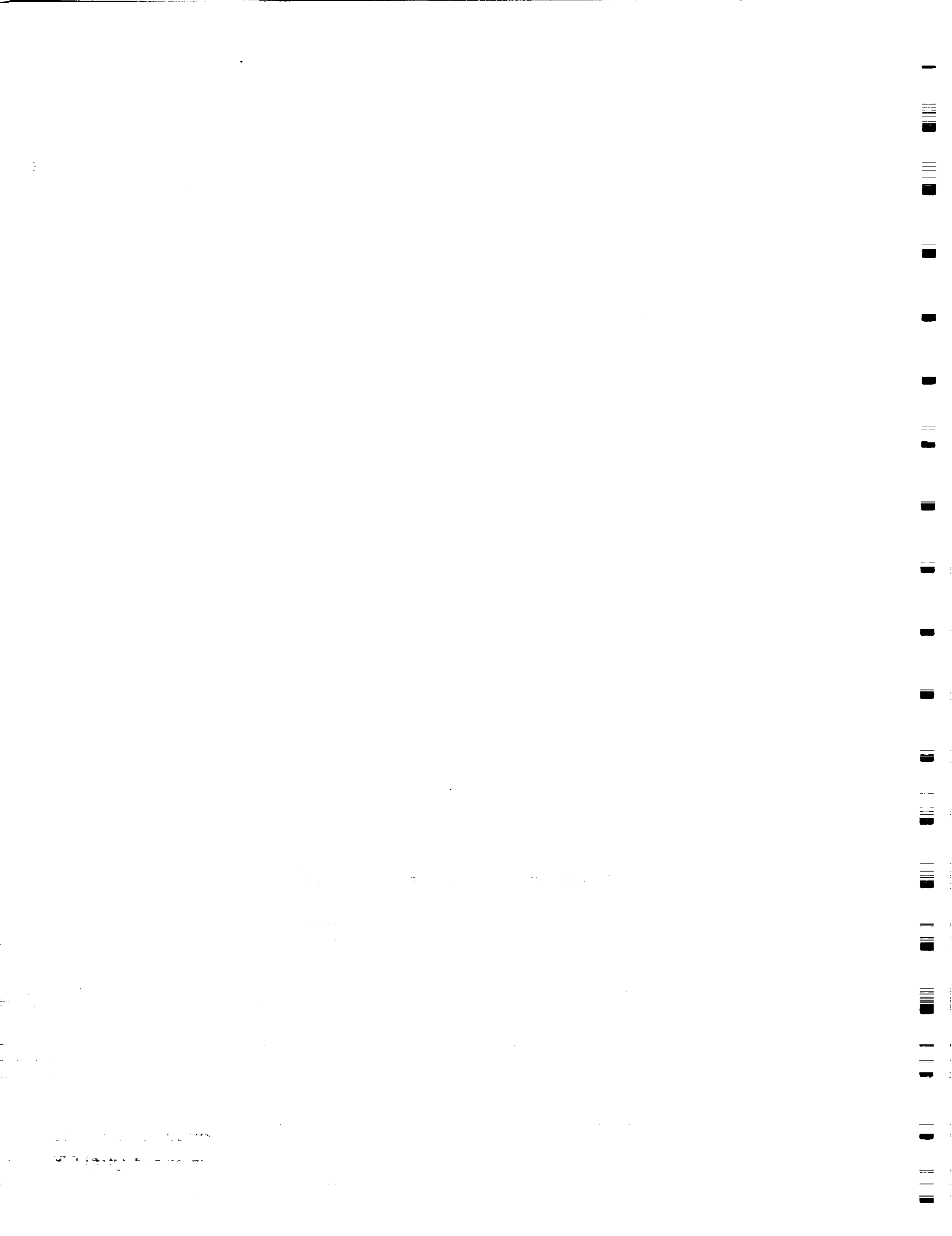
Avionics Systems Division, Research and Engineering
Johnson Space Center

Prepared by

SofTech, Inc.
in coordination with the
Research Institute
for
Computing and Information Systems
at the
University of Houston, Clear Lake

SofTech, Inc.

ORIGINAL PAGE IS
OF POOR QUALITY



VOLUME I

COMMENTS ON THE NETWORK COMMUNICATION SERVICES IN THE TINMAN USER'S MANUAL FOR DATA MANAGEMENT SYSTEM (DMS) TEST BED

SOFTECH

TABLE OF CONTENTS

Section		Page
1	INTRODUCTION	1-1
2	REVIEW OF POLICY STATEMENTS	2-1
3	REVIEW OF THE TINMAN DMS USER'S MANUAL	3-1
	3.1 Avoiding Active Polling	3-1
	3.2 The Type of Datagram Contents	3-2
	3.3 Avoid Use of Physical Addressing Schemes	3-3
	3.4 The Need for More Basic Services	3-3
	3.5 Scope of the DATAGRAM Generic Package	3-5
	3.6 Assembly-Language-Style Records	3-6
	3.7 Support for Multilevel Security	3-7

PRECEDING PAGE BLANK NOT FILMED

Section 1

INTRODUCTION

This volume provides a critical review of the network communication services contained in the Tinman User's Manual for Data Management System (DMS) Test Bed (hereinafter, referred to as the Tinman DMS User's Manual) prepared by the Lockheed Engineering and Management Services Company, Inc. under Contract NAS 9-15800, Job Order 34-208. The review is from the perspective of applying modern software engineering principles and using the Ada language effectively to ensure the test bed network communication services provide a robust capability.

Overall the material on network communication services reflects a reasonably good grasp of the Ada language. Language features are appropriately used for most services. Design alternatives are offered to provide improved system performance and a basis for better application software development.

Section 2 contains a review and suggested clarifications of the Statement of Policies and Services contained in Appendix B of the Tinman DMS User's Manual; Section 3 is a review of the Network Communication Services; and Section 4 contains concluding comments.

Section 2

REVIEW OF POLICY STATEMENTS

We are interested in clarifying the intent of certain statements of policy contained in Appendix B of the Tinman DMS User's Manual. For each such statement we provide a discussion below explaining why a clarification is needed and suggest how the statement might be rephased.

Statement #3:

The DMS design must isolate the subsystems so that malfunctions or changes to the software in one subsystem do not affect the operation of any other subsystem.

Discussion:

A subsystem may be built in such a way that it will not function correctly if it does not receive the data it is expecting from another subsystem. While the DMS can provide tools for building robust subsystems, the use made of these tools is beyond the control of the DMS. A change to one subsystem affecting the logical structure of its communication with another subsystem may indeed affect the operation of the other subsystem.

Recommendation:

Replace with:

The DMS design must isolate the subsystems so that:

a. changes to the software in one subsystem do not affect the operation of any other subsystem, provided that the changes preserve the

SOFTECH

logical communications channels of the first subsystem and the protocols for using each channel;

b. software malfunctions in one subsystem do not affect the operation of any other subsystem, except insofar as the second subsystem fails to receive data it was expecting from the first subsystem.

Statement #5:

The DMS will support real time communications that do not exceed specific transport delays (minimum and maximum). Subsystems requiring more capability must provide their own dedicated bus. The DCN component of the DMS will support a maximum (20ms.) single packet transfer time from point to point (subsystem to subsystem) within the same LAN. This is to include all overhead associated with message transfer through all seven ICY layers.

Discussion:

We assume there is no requirement for the application programmer to specify a maximum packet transfer time explicitly. For example, priorities can be associated with communications channels, with an implicit correspondence between priorities and transfer time.

Recommendation:

Replace the second sentence with:

The DMS may associate maximum transfer times with virtual circuit priority levels so that a subsystem can specify the maximum allowable circuit time by requesting a virtual circuit at a given priority level. A subsystem requiring a smaller transfer time than that associated with the highest priority level must provide its own dedicated bus.

Statement #6:

Near real time communications will be isolated from other types of data communications on the network (e.g., by unique identifiers, by multiplexing the available bandwidth into near real time services and non-near real time services, etc.). Connection establishment for near real time communications must allow bypassing of logical name to physical address translations and minimize network transport delays associated with connection establishment of human users.

Discussion:

There are two ways to interpret the sentence "Connection establishment for near real time and non-near real time communications must allow bypassing of logical name to physical address translations ....":

1. Establishment of the connection allows translation to be bypassed every time actual data is sent. (Translation may be done once at the time of connection, and the translated (physical) address is saved for use in routing data.)
2. It must be possible to bypass address translation in the actual establishment of the connection, that is, to establish a connection to a physical address.

We assume the first interpretation is intended.

Recommendation:

Replace the second sentence with:

When the DMS establishes a connection for near-real-time communications, it must provide the communicating subsystems with "handles" uniquely identifying the connection. During establishment of the connection, logical network addresses may have to be translated to physical addresses, but the "handle" will allow subsequent access to the communication channel without further translation.

Statement #7:

It must be possible to prioritize messages so that messages with higher priority are handled first. A receiving subsystem must have the ability to examine the incoming message queue to determine each message's attributes (e.g., priority, origin, relative position in the queue, time tag, time out value and size). It should be possible for a subsystem to remove a message from the incoming queue or change its priority.

Discussion:

This policy statement implies the existence of certain requirements that are not stated explicitly elsewhere, including the routine logging of time tags, sender's identification, and message size; and the ability to specify a time-out value when sending a message. Are these indeed requirements, or just examples of the kind of attributes an incoming message might have? Must these capabilities be provided for all messages?

Recommendation:

Delete the second and third sentences.

Statement #11:

Data will only be transmitted in response to a request. In general, subsystems will not broadcast data or periodically send data unannounced to another subsystem. Exceptions to this policy will only be allowed in emergency situations.

Discussion:

The crux of this policy statement is that a subsystem should only receive data it anticipates receiving. In particular:

It may sometimes be reasonable to broadcast a particular kind of message to a set of subsystems all designed to handle that kind of message. This capability must be supported even if the principle of isolating subsystems dictates that it be used sparingly.

Even in emergency situations, messages should only be broadcast to those subsystems designed to look for the messages and respond sensibly, even if the only action is to disregard the message. A subsystem expecting only attitude readings on a certain channel will not recognize a loss-of-pressure alarm sent on that channel.

Recommendation:

Replace with:

Data will only be transmitted to subsystems designed to receive and handle the data. (This handling may consist of recognizing and then ignoring certain kinds of messages.) Transmission of a given kind of data to a given subsystem will be allowed only if explicitly permitted by that subsystem's ICD.

Statement #12:

Subsystems communicate with other subsystems, not with processes within a subsystem. If a process in subsystem A needs information from a process in subsystem B, it addresses its request to subsystem B who is then responsible for determining which process will respond to the request.

The NOS will route messages to subsystems, not processes within a subsystem. The intent is that if subsystem A needs information from a process in subsystem B it addresses its request to subsystem B who is then responsible for determining which process will respond to the request. However, in the event that the need for direct process to process message routing is identified, the following option will be provided. Subsystem A may optionally append the name of the process in subsystem B (to which the message is to be routed) to B's address. The NOS will still route to subsystem B but subsystem B will use the appended process name to place the message in the proper incoming queue without the necessity of determining which process is to service the request.

Discussion:

Irrespective of its internal structure, a subsystem may have several logical input and output streams (communications channels). When we say that the NOS routes messages to a subsystem, we mean that the NOS routes messages to particular logical input streams that are part of the subsystem's interface. Internally, a subsystem may delegate the task of establishing and using communications channels to particular processes. The practical effect is that messages are routed directly to the processes that handle them, but the abstract view is that the receiving subsystem has one or more logical input streams and that the message was routed to one of these. This view neither precludes or requires a subsystem design in which there is one logical input stream and a central dispatching process that passes a message along to a particular task based on its contents.

Recommendation:

Replace with:

Subsystems request connections to other subsystems, not to processes within another subsystem. However, messages may be sent and received on

SOFTECH

behalf of a subsystem by particular processes within that subsystem. This results, in effect, in direct process-to-process communication even though each subsystem is oblivious to the internal structure of the other subsystem. One process within a subsystem may route a message to another process in the same subsystem based on the contents of the message, but this detail about the internal implementation of the subsystem is irrelevant to other subsystems.

Statement #13:

By adhering to the intent of policy statement 12 above the structure of a subsystem can be hidden from the BIU and from human and software users. Changes to the internal structure of a subsystem, especially those involving the removal or renaming of processes will then be transparent to human and software users. Furthermore, it will allow the NOS to maintain configuration data at the subsystem level rather than the process level. This policy can be stated more generally as: It is the responsibility of each subsystem to manage its own resources, including the determination of which process will service which request.

Discussion:

The approach suggested for policy statement #12 supports the fulfillment of policy statement #13. The logical input and output streams of a subsystem must be distinguished from the internal structure. Logical input and output streams are part of a subsystem's interface, but its process structure is an internal implementation concern. Processes can be added or removed at will, as long as some process continues to handle each logical stream, but addition or removal of a potential stream connection is an external change that may affect other subsystems. (Virtual circuits may be created and removed dynamically during subsystem execution, but only if the logical input and output streams to which they are connected are part of the subsystem's interface.)

Recommendation:

Add at end:

(Addition or removal of processes must be distinguished from addition or removal of potential communication channels from a subsystem's ICD. While the former is an internal change to a subsystem, the latter is external and may affect other subsystems.)

Statement #14:

When subsystem A attempts to establish a connection to another subsystem B, B must respond to A before it can be assumed that a connection exists.

Discussion:

We assume it is not necessary for B to respond directly and explicitly to A. For example, the NOS routine called by A to establish a connection might wait for a response from B before returning a circuit ID to A. This ensures that A cannot do anything until a connection has been established. From A's point of view, the effect of the call is simply to obtain a circuit ID, and the communication with B is hidden.

Recommendation:

Replace with:

When subsystem A attempts to establish a connection with subsystem B, B must also inform the NOS of its readiness to establish a connection before A is permitted to use the connection.

Statement #28:

Each subsystem should have a mnemonic name. Remote users (human and subsystem) should be able to address messages to the "ECLSS" or "GN&C" subsystems regardless of where they are physically located. Each LAN and each region should have a mnemonic identifier so communications from processes in other LANs can logically address transmissions across the network. The DCN will translate these logical names into and from unique physical addresses.

Discussion:

Policy statements 26 and 27 wisely call for hiding the physical location of network resources from users, and the requirement in policy statement 28 to be able to name subsystems by mnemonic names further supports this policy. Besides simplifying the user interface, this approach provides flexibility to change the physical network configuration without affecting subsystems.

However, we do not understand the need for a name consisting of a region mnemonic and a LAN mnemonic. Such a name is not a logical name, but a physical name, specifying the location of a subsystem within the network. A true logical name uniquely identifies a subsystem within the network without reference to the LAN containing the subsystem or the region containing the LAN. Indeed this allows different parts of the same subsystem to be split across the DMS network.

Recommendation:

Remove the third sentence.

SOFTECH

Statement #33:

The design of the IOC DMS will be based on common subsystem and BIU processor types. However, the design should in way require a common processor type. It should be sufficient that they have the same external interfaces and obey common communications protocols. Also, the design of the IOC DMS must accommodate communications with heterogeneous LAN types within regions to include heterogeneous gateways. The intent is to maximize commonalty without restricting flexibility to upgrade technology.

Discussion:

We presume the second sentence is meant to read "However, the design should in no way require a common processor type."

Recommendation:

Add the word "no" to the second sentence:

However, the design should in no way require a common processor type.

Statement #35:

Data types, i.e., machine representations (including floating and fixed point and bit and byte ordering) will be pulled from the Federal Standards in order to limit the number of and clearly state the representations accepted for layer 6. Data format conversion is not a DMS function.

Discussion:

We are concerned about the reference to "limiting" the representations accepted for layer 6. Ada supports the definition of new problem-oriented data types, and it will often be useful to transmit objects in these data types from one subsystem to another. However, it is impossible to

enumerate in advance all data types and to specify a representation for each. Resolution of this problem requires stepping back from the DMS requirements and considering requirements for the compilers that will be used to compile subsystems.

A standard binary representation for transmission between subsystems is indeed needed. It would be wasteful of processor time and bandwidth to convert all complex data into textual form for transmission. It would be inconsistent with the software engineering principles of data abstraction and information-hiding and the intended use of Ada, and inimical to program reliability and maintainability, to require application programs to decompose abstract data into its constituent elements and transmit these elements individually.

Rather, standard representations can be provided for scalar data types (numeric and enumeration types), and standard rules can be provided for constructing the representation of composite types (array and record types) from the representation of their components. (Example: "The representation of a record consists of the representations of the record components existing in that record, in order of declaration, with each component starting on a byte boundary; arrays are stored in ascending order with the last index varying most rapidly and the representation of each component beginning on a byte boundary.") In addition to scalar and composite types, the Ada language has access types, task types, and private types. The transmission of access-type and task-type values from one Ada main program to another is generally not meaningful and should not be supported. Each private types has an underlying representation as a scalar, composite, access, or task type.

In the spirit of policy statements 2 and 35, it is not necessary to restrict Space Station software developers to Ada compilers that always use the standard representations internally. If necessary, the Ada runtime system can perform the necessary conversions on incoming and outgoing data. Alternatively, to ensure that conversion can always

SOFTECH

be bypassed, NASA might impose a less stringent requirement: that compilers must support representation specifications to a sufficient degree that a programmer can stipulate the network standard representation for objects of a particular type. (Representation specifications are a low-level feature of the Ada language that control the internal representations for programmer-defined data types. Normally, a compiler may choose which representation specifications to obey and which to reject, so not all representation specifications are recognized by all compilers. Certain matters of internal representation, such as the order in which array components are stored, are beyond the control of representation clauses. Compilers must either conform by default to the network standard in such matters or else provide some means, such as an implementation-defined pragma, for the programmer to request conformance in particular cases.)

Whenever a high-level, abstract data type (e.g., a type for representing celestial coordinates) is of interest to more than one application, we would expect the type to be defined in a package residing in a network-wide library. Individual subsystems (i.e., Ada main programs) can import the type definition by a with clause for that package. Representation specifications for the type, located in the importing package, will ensure that all subsystems using the type have a common representation for it, so internal representations can be transmitted from one importing subsystem to another without conversion. Network standards would ensure that these representation specifications are accepted by the compilers used for each subsystem.

Recommendation:

Remove the words "limit the number of and" from the first sentence.

Section 3

REVIEW OF THE TINMAN DMS USER'S MANUAL

The following sections identify the major areas of concern in the Tinman design. A discussion of each area is provided with recommended changes.

3.1 Avoiding Active Polling

The datagram facilities require active polling for incoming messages. This method of communication is efficient for systems in which data is expected to arrive at a rate that is of the same order of magnitude as the polling rate. However, if the average message arrival rate is much lower than the polling rate then the process is inefficient and uses more computer resources than necessary. Also of concern is the impact the polling model has on the design of the application software that uses network communication services. The polling model has its basis in sequential processing languages and encourages a sequential style of design for applications, even when the application has naturally occurring concurrency. This in turn results in an application design that is difficult to modify.

A better model for communication is one that readily supports both sequential and concurrent design styles. The Tinman model could support both styles if the RETRIEVE call were to wait for up to some specified maximum period for an incoming datagram. If a datagram were to arrive within the period, then it would be returned; otherwise a time-out exception would be raised. The retrieving process would be blocked during the time-out period. If the receiving subsystem had useful work to do while waiting for an incoming datagram, it could use Ada's powerful task synchronization primitives. The task calling RETRIEVE could be blocked while another task in the subsystem continued to execute. The time-out period associated with the RETRIEVE could be used to either avoid a deadlock situation or to return immediately for highly time-critical applications that did not use tasking. A time-out period

of zero seconds would behave like the polling model. Thus both sequential and concurrent style designs could be easily achieved.

3.2 The Type of Datagram Contents

The Tinman manual anticipates that datagrams of different types will all arrive in the same incoming queue. By allowing all datagrams to arrive on the same queue modifications that do not change the logical communications interface can be made to a subsystem without affecting other subsystems. Different instantiations of the generic package DATAGRAM will provide versions of RETRIEVE for receiving datagrams of different types. In some respects, this is a very sensible approach. The Ada language encourages the definition of new abstract data types to model application entities, and it is reasonable to expect that applications will communicate by passing values of these abstract data types as messages. An unlimited number and variety of application-oriented data types are possible, so there is no way for the NOS specification to anticipate all possible message types in advance.

In general, it is difficult and possibly inefficient to manage any queue unless the messages in the queue are of a fixed type (in some cases a record type with a fixed number of variants). If the messages are of different types then they must first be examined to determine their type before the appropriate instantiation of the RETRIEVE can be called and an appropriate routine can be invoked to handle the message. As the number of message types increases, the logic involved in this decision becomes more complex, increasing the likelihood of error. Also, if the subsystem is distributed across the DMS network, the retrieving process may be located on a different node than the routine that handles the message. In this case, the retrieving process must then send another message across the network, increasing network traffic.

For improved modularity it is more appropriate to have distinct queues for each type of message. The receiving subsystem should expect incoming messages

of specific types on specific queues. Since the queues are typed, it is unnecessary to first examine the message to determine the correct instantiation of RETRIEVE and a central dispatching routine is not required. Also, in a distributed subsystem, messages can be routed by the NOS directly to the network node which supports the function that handles the message. Changes to the subsystem can still be made without affecting other subsystems as long as the logical communications interface requirements have not changed.

3.3 Avoid Use of Physical Addressing Schemes

As noted earlier in the discussion of policy statement 28, it is unwise to designate a subsystem address in terms of the region and LAN in which it is physically located. The definition of ADDRESS_TYPE for a subsystem address includes the physical location of the subsystem. This approach makes restrictive assumptions that may prevent the distribution of a subsystem over more than one LAN or region if this eventually becomes feasible. It will also require massive reprogramming every time the physical location of a subsystem changes, since all references to the subsystem address must be changed. A wiser approach is to give each subsystem a logical name that is translated by the NOS into a physical location, based on tables maintained by a network administrator.

3.4 The Need for More Basic Services

The Tinman design provides a number of powerful facilities that may be needed for certain applications, including the ability to scan the incoming message queue and the automatic establishment of a bi-directional connection every time a connection is established. However, this design may impose an unwanted overhead on applications that do not require such sophisticated capabilities. A better approach is to provide basic NOS services that are simple and efficient and can serve as building blocks for implementing more sophisticated services, rather than to try to anticipate the complex

combinations of features that some application writers might find useful. As subsystem communication requirements crystallize, utilities can be written on top of the basic network services to provide commonly needed higher-level capabilities.

Consider the SCAN command, for example. We have doubts about the wisdom of including such sophisticated queue-manipulation and message-receipt-scheduling operations as basic network communication services. Here are some reasons:

1. We expect this functionality will be required by few subsystems, but it will add to the complexity and overhead of network communications even for subsystems that do not require such functionality. Much of the time, for example, one subsystem will expect to receive messages from another particular subsystem, so it will be wasteful to have the sender and receiver identified in each message. Likewise, few applications would require all the attributes described (such as CLASS, TIME_TAG, or STATUS) to be sent with every message.

2. Sophisticated manipulation of the incoming message queue and extraction of higher priority messages are presumably aimed at the timely processing of important messages, but we expect examination, analysis, and manipulation of the queue to itself be quite time-consuming. In most cases, a subsystem is more likely to provide timely service by quickly removing items from a queue in order of arrival rather than by trying to schedule the handling of enqueued messages. The effect of messages with different priorities can be achieved by the provision of multiple communication channels with different priorities. The receiving subsystem would always look for messages on high-priority channels first.

3. For subsystems that must service messages in some order other than order of arrival, queue manipulation can be provided by the subsystem itself (perhaps with the aid of general-purpose NOS utility that is not part of the network communication services). Such a subsystem would have a process whose sole responsibility is to remove messages from an incoming queue and insert them in a subsystem-internal data structure as quickly as possible. The subsystem would have complete control over the data structure, including the removal of messages from the data structure so they can be processed.

4. Ada's powerful data abstraction facilities allow the structure of a message to be arbitrarily complex. If the application requires it, information like the time of transmission, the identity of the sender and other attributes can be included in the message.

Similarly, consider the automatic establishment, every time a virtual circuit is established to connect two subsystems, of incoming and outgoing queues for both subsystems. By default, subsystems should be able to request a virtual circuit for communications in one direction. In many cases this may be all that is required. Given such a capability, it is easy to implement bi-directional communications.

3.5 Scope of the DATAGRAM Generic Package

Each instantiation of a generic package creates a new and distinct instance of every entity provided by the generic package. The point may be moot given the message-type problem identified earlier, but some of the facilities currently provided by instances of DATAGRAM should not be declared in a generic template. It is logically necessary for each instantiation to produce a new DATAGRAM_TYPE and new versions of SEND and RETRIEVE. However, it would make sense for there to be only one MESSAGE_COUNT type for use by all instances of the generic package, and one HEADER_TYPE (so that a common set of header-manipulating utilities, applicable to all types of datagrams, could be written). Similarly, new ADDRESS_BLOCK and ADDRESS_LINK types, and new MULTICAST and DELETE_MN procedures, should not be created for each datagram type. (It is not clear whether the declaration of TIME_STAMP is a subtype declaration with "type" inadvertently written instead of subtype, in which case the same subtype is shared by all instances of the package, or a derived type declaration with the word "new" inadvertently omitted, in which case each instance provides a distinct TIME_STAMP type. Logically, a single type is more appropriate.) There should be one set of exceptions raised by subprograms in all instances of the package, not distinct exceptions for each instance.

One solution to this problem is to make DATAGRAM a nongeneric package and nest a generic package inside of it. Only entities like DATAGRAM_TYPE, SEND, and RETRIEVE would be declared in the generic package. Another solution is to declare in a separate package, say DATAGRAM_TYPES_PACKAGE, the entities to be shared by all instances of the generic package DATAGRAM. DATAGRAM would be given a with clause for DATAGRAM_TYPES_PACKAGE and could also include renaming declarations for the entities provided by DATAGRAM_TYPES_PACKAGE. This would make it appear that all entities were being provided by each instance of DATAGRAM, but an entity declared by renaming declarations would be a single entity, created once and passed along by many instances.

It is not clear whether SCAN should be provided by the generic package or made common to all instances. The parameter and result types of SCAN are the same for all instances. However, if we view each instance of the generic package as creating a distinct queue for datagrams of a particular type, each instance could provide versions of SCAN for examining that particular instance's queue.

3.6 Assembly-Language-Style Records

The HEADER_TYPE record component, contained in the DATAGRAM_TYPE record, is used as an assembly-language-style control block. Different subcomponents are set and examined by different modules at different times, and some subcomponents, particularly the STATUS component, have multiple uses. Data flow is obscure and complex, modules using the records become more tightly coupled than is necessary or desirable, and error-prone protocols are imposed on each subsystem. HEADER_TYPE records should be used only within the NOS, and hidden from users of the NOS. Information to be provided by the sender to the NOS should be provided through separate in parameters to SEND. Information to be provided by the NOS to the sender should provided through out parameters to SEND. Similarly, specific in and out parameters to RETRIEVE should be used to convey information from the receiver to the NOS and from the NOS to the receiver, respectively. It is appropriate to group items in a

record when the items can be taken together as modeling a single abstract entity, but the components of HEADER_TYPE records do not meet this criterion.

3.7 Support for Multilevel Security

The Tinman user's manual does not address the multilevel security issues raised by Policy Statement 30. The implications of multilevel security requirements for network communications require further study. For example, there may be restrictions on the ability of two subsystems to establish a connection in a particular direction based on the security level of each subsystem. This might be reflected in an exception SECURITY_VIOLATION that can be raised by the connection-request subprograms. It may also be necessary to establish virtual circuits with different security levels, implemented by different kinds of physical connections.

Section 4

CONCLUSIONS

The Tinman User's Manual reflects much careful thought about the problem of Space Station Network Communications. Our major concerns are confined to three areas:

1. The requirement for active polling of incoming messages. The active polling requires excessive resources if the rate of arriving messages is significantly less than the polling rate. This is particularly problematic if the computing resources are required to perform a significant amount of other work. Active polling complicates the design of applications with naturally occurring concurrent processing. The use of communication design structures that are compatible with application structures reduces the complexity of the design of the application, thus reducing the cost of developing and maintaining the application software. The requirement for active polling also makes the network communication services less flexible to the users of those services.

2. The use of single data queues or virtual-circuit connections for messages of different types. The use of single datagram queues or virtual circuit connections may have significant impact if the subsystem is distributed on the DMS network, since messages received by the subsystem may have to be retransmitted across the network to the appropriate process to handle that message type. Furthermore, the use of a single datagram queue or virtual circuit connection for messages of different types does not provide a rational structuring of information. It forces the application programmers to manage messages of different types. Again, the network communication services are made less flexible for users.

3. The remaining hints of physical network configuration in the structure of logical addresses. Basing the services on a physical model of the DMS network significantly reduces the flexibility of system designers, particularly if a subsystem must move to different physical parts of the network due to design changes or dynamic reconfiguration. Failure to hide network topology and separate the concerns of logical and physical viewpoints, will lead to massive reprogramming if the physical location of a subsystem changes.

VOLUME II

DESIGN ALTERNATIVES FOR THE NETWORK COMMUNICATIONS SERVICE IN THE DATA MANAGEMENT SYSTEM (DMS) TESTBED

SOFTECH

TABLE OF CONTENTS

Section		Page
1	INTRODUCTION	1-1
2	NETWORK MODEL	2-1
	2.1 Physical Network Model	2-1
	2.2 Logical Network Model	2-3
	2.3 Mapping of the Logical model Onto the Physical Model	2-5
	2.4 Users of the NOS	2-7
3	REQUIREMENTS	3-1
4	SERVICES	4-1
	4.1 Overview	4-1
	4.1.1 Virtual-Circuit Communication	4-3
	4.1.2 Datagram Communication	4-5
	4.1.3 Broadcast and Multicast Services	4-6
	4.1.4 The Role of Tasking	4-10
	4.2 Specification of Network IO	4-11
	4.3 Behavior of Network IO Subprograms	4-12
	4.3.1 Open	4-12
	4.3.2 Close	4-14
	4.3.3 Datagram Output File	4-14
	4.3.4 Datagram Input File	4-15
	4.3.5 Set Data Unavailable Response	4-15
	4.3.6 Read	4-16
	4.3.7 Write	4-16
	4.3.8 End of File	4-17
	4.4 Summary of Exceptions	4-18
	4.4.1 Data Unavailable	4-18
	4.4.2 Status Error	4-19
	4.4.3 Mode Error	4-19
	4.4.4 Name Error	4-19
	4.4.5 Use Error	4-19
	4.4.6 Device Error	4-20
	4.4.7 End Error	4-20
	4.4.8 Data Error	4-20

PRECEDING PAGE BLANK NOT FILMED

TABLE OF CONTENTS (CONT.)

Section		Page
5	EXAMPLES OF USE	5-1
	5.1 Processing a Bounded Sequence of Data	5-3
	5.2 Performing Background Processing While Waiting for Datagrams	5-4
	5.3 A File Server	5-10
	5.4 Converting Arriving Messages to Entry Calls	5-12
	5.5 Merging Streams of Incoming Messages	5-16
	5.6 Processing Datagrams of Different Priority	5-22
	5.7 Using Message Contents to Control Order of Processing	5-26
	5.8 Using Datagrams to Control Periodic Sampling	5-31

LIST OF ILLUSTRATIONS

Figure		Page
2-1	Physical Network Model	2-2
2-2	Logical Network Structure of Processing Entities	2-3
2-3	Visibility Between Processing Entities	2-5
2-4	Mapping of Processing Entities to Ada Programs or Tasks	2-6
4-1	Calls on Open Waiting Normally Until Both Parties Have Requested a Connection and the Connection Has Been Established	4-2
4-2	Behavior of the End_of_File Function	4-18
5-1	Notation for Network Communication Patterns	5-2
5-2	Notation for Program Structure	5-2
5-3	Flow of Messages from PAYLOAD_DATA_MANAGER to DATA_ANALYZER	5-3
5-4	Flow of Requests for Data and Averaged Sensor Readings	5-5
5-5	Structure of the Program to Provide Average Reading on Request	5-7
5-6	Flow of Messages Between FILE_SERVER and Client	5-10
5-7	Program structure using Virtual_Circuit_Delivery_Template	5-15
5-8	Program structure using Datagram_Delivery_Template	5-17
5-9	Flow of MESSAGES to and from POWER_SUPPLY_MONITOR	5-18
5-10	Program-unit structure of POWER_SUPPLY_MONITOR	5-19
5-11	A Processing Entity with Datagram Streams Corresponding to Different Priority Levels	5-22
5-12	Task structure for processing bulletins of different priorities	5-24
5-13	Use of the Warning_Retriever Task to Control the Order in which Messages are Processed	5-29
5-14	Network communication with SENSOR_MANAGER	5-32
5-15	Program-unit structure for SENSOR_MANAGER	5-38

Use or disclosure of proposal data is subject to the restriction on the Title page of this Proposal

ORIGINAL PAGE IS OF POOR QUALITY

Section 1

INTRODUCTION

The Space Station Data Management System (DMS) provides a network operating system (NOS) for communication services between network users. This report describes a proposed network model and NOS communication services. Since the Space Station has selected Ada as the programming language for DMS software, the proposed NOS services build upon conceptual I/O models of Ada for a parsimonious design. This approach leads to simpler services that can be used in a straightforward way.

The set of NOS communication services contained in this report represents an interface between the application layer (layer 7) and the presentation layer (layer 6) of the Open Systems Interconnection (OSI) Basic Reference Model. It is not intended as an interface set for a typical DMS user as not all DMS users will be knowledgeable in Ada, but as one of several NOS interface sets available to an Ada programmer who is developing applications that use the NOS. Other NOS interface sets, such as virtual terminal, file services etc. and DMS user interface sets are not addressed herein.

Section 2 of the report presents the proposed network model of the DMS. Section 3 describes the functional requirements of the NOS communication services; Section 4 provides the services that represent the application programmer's interface to the NOS communication services; Section 5 includes some example applications illustrating a wide range of flexible communication capabilities.

Section 2

NETWORK MODEL

The DMS network is represented as both a physical network and a logical network. The physical network represents the actual hardware to support program execution and communication. The topology of the network is not significant in the discussion of the NOS but an aid in understanding network terminology. The logical network represents the processing entities and their relationships to support the mission requirements. A clear distinction between these two views and an appropriate way to relate them to each other is presented below. The kinds of NOS users are also defined.

2.1 Physical Network Model

The DMS network physical model is a hierarchical structure consisting of a set of regions, each of which consists of a set of local area networks (LAN), each of which consists of a set of nodes. The nodes are the physically addressable units in the network. A region may be considered as a single Space Station or ground control center. Regions are generally geographically collocated collections of LAN's. Regions will typically access each other with telecommunications, while LAN's within a region will typically have hardwired access to each other.

Nodes may be implemented as a network interface unit (NIU) and a set of processors and other equipment (e.g., sensors, control units, peripherals, etc.). Processors which must access the network communicate through the NIU. Communication between regions and LAN's is supported by network bridges that may be implemented as special purpose NIU's. Each copy of the NOS will run the same on different nodes. A node may contain one or more processors, but this is hidden from the NOS. A special node executive (NE) may be required to present a consistent processor model to the NOS.

Processors are not directly addressable on the DMS network. All DMS network communications must be between nodes on the network. This does not preclude the use of additional or special purpose networks that may exist between processors and other equipment. Direct processor-to-processor communication may use a variety of methods, but it does not use the DMS network. Such communication is beyond the scope of the NOS. Figure 2-1 illustrates the physical model of the DMS network.

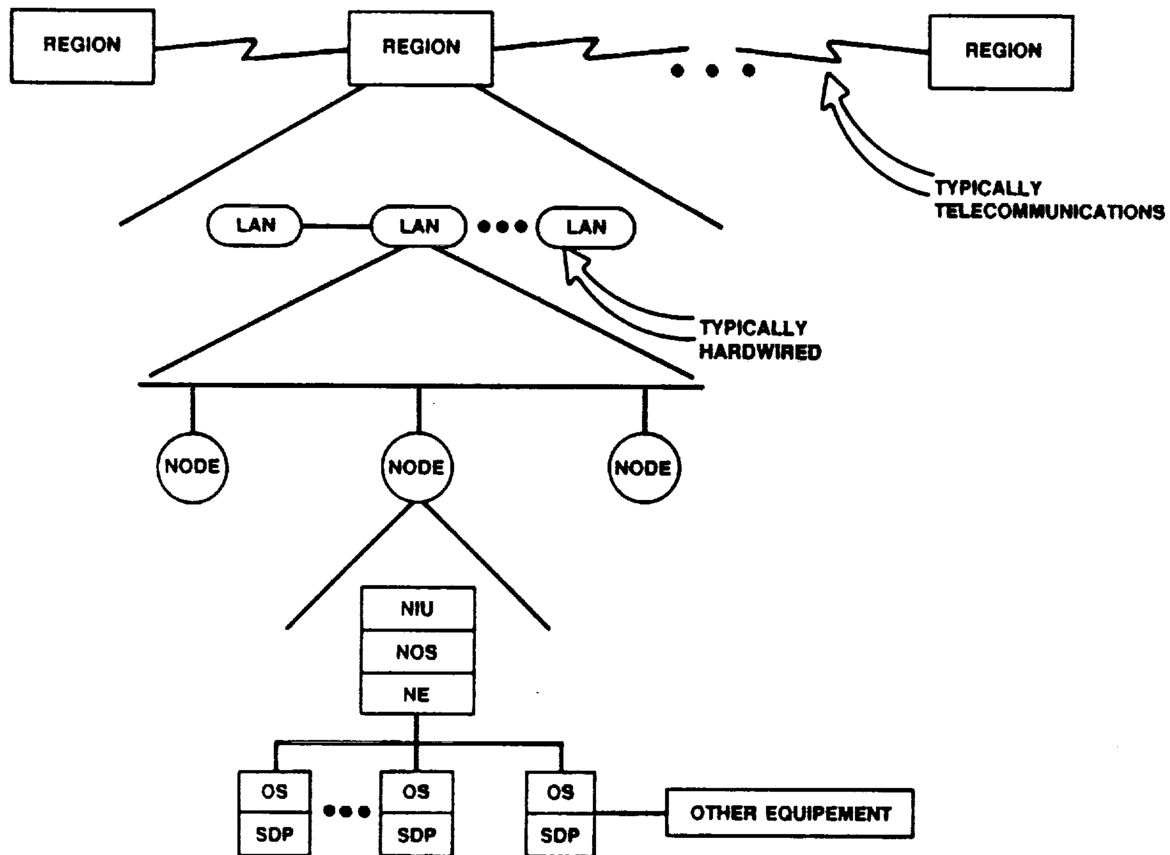


Figure 2-1. Physical Network Model

2.2 Logical Network Model

The logical model of the DMS network is not fully defined. Notions of subsystems and processes do exist, but the explicit logical structure and communication capabilities within and between subsystems are still evolving. The following is a proposed logical network model that serves to reduce the complexity of the software for the Space Station Program. While the model is recommended as an approach to system design, the specific services described in Section 4 later do not require it. It is presented to provide a framework for discussions within this report and as a concept for consideration within the Space Station Program community.

It is proposed the logical model of the DMS network be a hierarchical structure consisting of groups of processing entities with logical addresses that can communicate with certain restrictions. A Space Station processing entity, such as the GN&C or Flight Control subsystem, may in turn contain a group of entities that are its children. Subsystem entities may have parent system entities. These system entities represent groups of subsystems with common attributes. Likewise, the child entities of a subsystem may in turn be parents of other subgroup processing entities. Children of the same parent are siblings of each other. Figure 2-2 illustrates the hierarchy of processing entities.

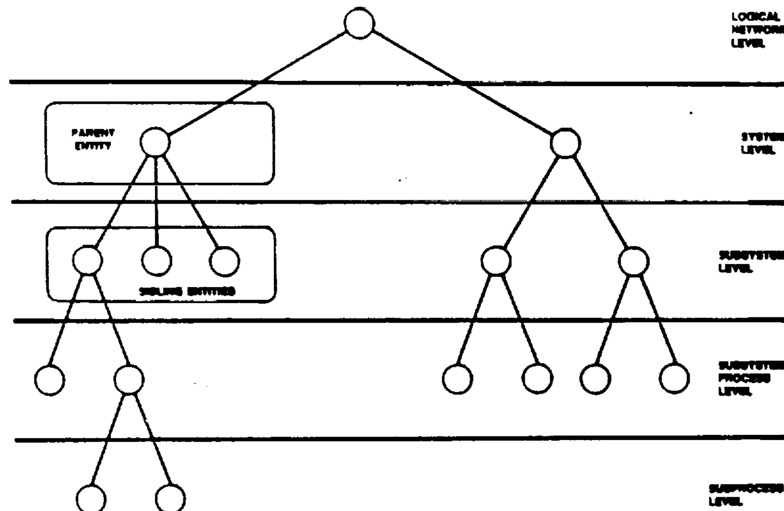


Figure 2-2. Logical Network Structure of Processing Entities

SOFTECH

One processing entity may communicate with another if it has access to its logical name. Naming of entities is, however, restricted in accordance with a visibility rule. The rule states a processing entity may only name its ancestors and their siblings. (This rule is analogous to the scoping rules of nested procedures in Ada.) Figure 2-3 depicts the visibility rule for entity names. In the figure, each circle represents a processing entity, and decomposition (children) is illustrated with circles inside circles. If the surface of each circle is viewed as a one-way mirror allowing an entity to "see" out of its circle then it can also "see" out of its ancestors' circles and "see" the siblings of its ancestors. "Seeing" the entities implies the logical name of the entity is accessible. However, an entity cannot see inside another's circle. The inner structure of an entity is hidden from other entities.

This model does not necessarily preclude communication between two arbitrary processing entities. While entities within two different subsystems can not directly "see" each other, they may nevertheless need to communicate. If entity A1 in subsystem A needs to send a datagram to entity B1 in subsystem B, then entity A1 would address the datagram to subsystem B. Subsystem B is then responsible to ensure the datagram is properly routed to entity B1. This allows the internal structure of subsystem B to change, both logically and physically on the network, without necessitating change to subsystem A as long as the logical interfaces between the two subsystems did not change (i.e. the specific function of entity B1 expected by entity A1 remains unchanged in accordance with policy #3 in Appendix B of the Tinman DMS User's Manual).

This model reduces the complexity of the software by reducing the number of visible interfaces within and between subsystems. It follows the principle of information hiding which is a major concept of modern software engineering. Software designs that adhere to this model result in systems that are significantly less costly to modify as compared with more traditional flat designs where every processing entity can, in principle, "see" every other entity.

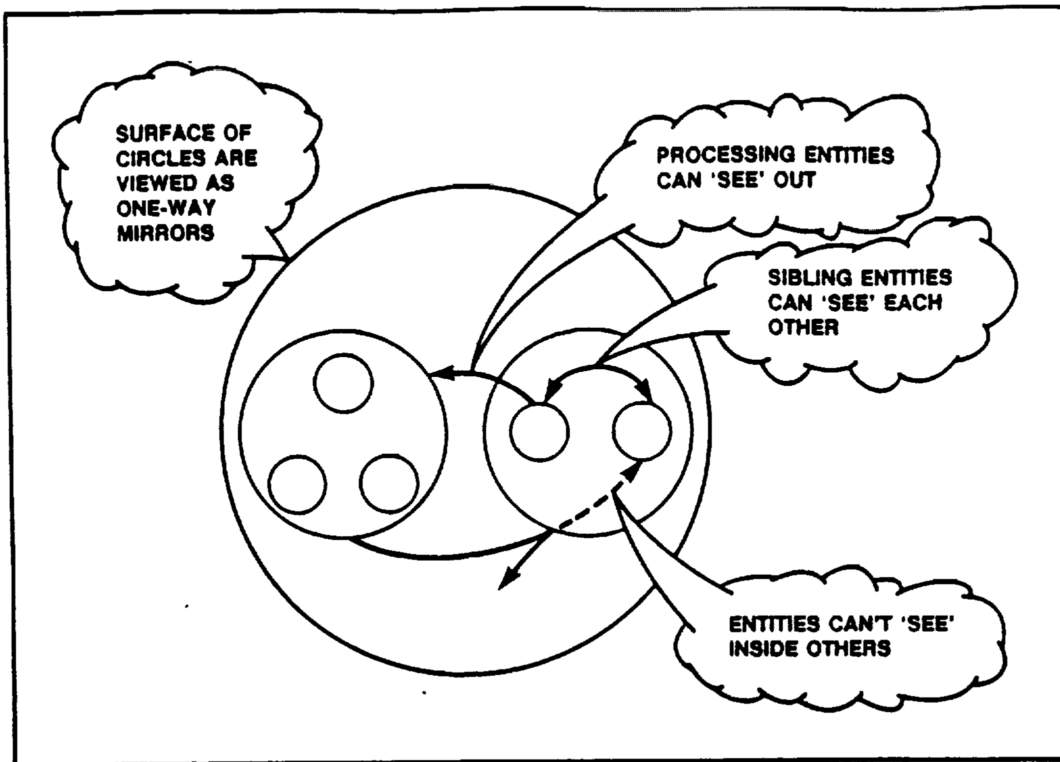


Figure 2-3. Visibility Between Processing Entities

2.3 Mapping of the Logical Model Onto the Physical Model

The mechanism for the mapping of logical processing entities, such as subsystems, onto physical processors that use the DMS network is the Ada program and the NOS interfaces.

Processing entities, such as subsystems, are supported by a set of Ada programs and/or tasks that call upon the NOS for communications services. The NOS is responsible for maintaining the configuration of the system (including the correspondence between the addresses of processing entities and the addresses of physical nodes) and communication among different nodes in the physical network.

An effective mapping of processing entities to Ada programs and tasks can aid in minimizing the cost of evolution, while a poor mapping may exasperate the process. One recommended approach during the design process is that designers map the logical model of processing entities in terms of the physical model so that an Ada program represents one or more sibling entities. Entities with different ancestors should not be grouped into the same Ada program. The lowest level descendent processing entities should be represented as tasks or main programs. Such a mapping is illustrated in Figure 2-4. This type of mapping reduces the coupling between processing entities and the side effects during evolution.

The way sibling entities are allocated to programs is influenced by a number of factors including, but not limited to, the physical distribution of entities across the network. Sibling entities that reside on different nodes will be allocated to different programs unless a distributed Ada program capability is available (i.e. tasks of a single program reside on different network nodes). Sibling entities residing on the same node can be allocated to the same program.

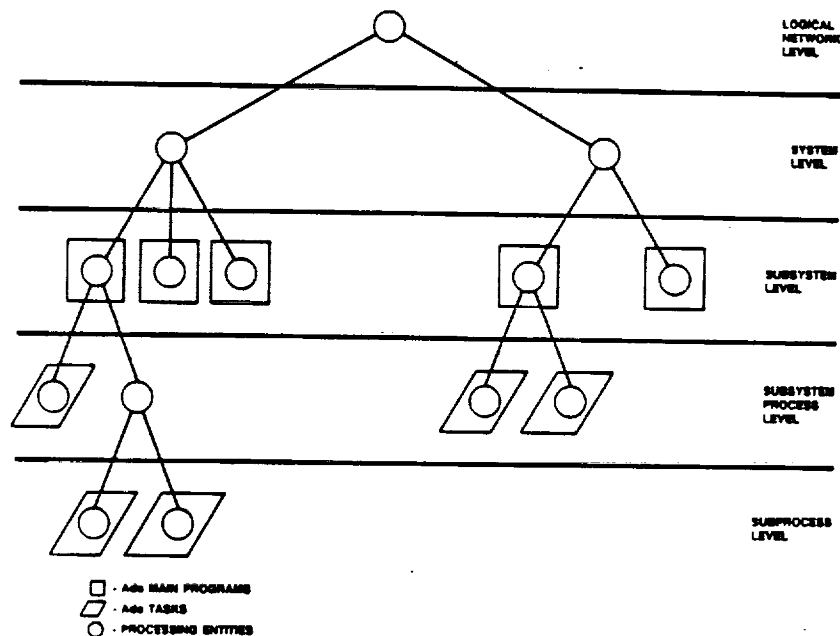


Figure 2-4. Mapping of Processing Entities to Ada Programs or Tasks

Designers should take care when allocating processing entities to a network node. If the node only supports a single Ada program then all entities should be siblings. Nonsibling entities may be allocated to a node if it supports multiprogramming since the entities may be mapped to different programs.

It is emphasized that this recommended mapping of processing entities onto the DMS network is performed during the software design phase and is not required by the NOS proposed later in this report.

2.4 Users of the NOS

Processing entities communicate with each other using two kinds of mechanisms. Entities mapped to different Ada programs will make explicit calls upon the NOS at the Ada source-code level to communicate with each other. The applications programmer is explicitly aware of this mapping and makes use of the NOS services described later. These services constitute the interface for the applications programmer. Thus, an Ada program is one type of NOS user.

Entities mapped onto Ada tasks within the same program rendezvous to effect communication. Thus, the second potential user of the NOS is the Ada Run Time System (ARTS). The ARTS is responsible for communication among tasks of an Ada program. If the single Ada program is distributed across the DMS network, then the ARTS may require the NOS to effect the communication between Ada tasks. Such use is implicit at the Ada source code level, since the NOS calls are made by ARTS rather than explicitly in the Ada program. The set of NOS interfaces for the ARTS may be different from those described in this report.

In the strictest sense only a few processing entities are direct users of the NOS. These may be viewed as agents which act on the behalf of other entities for communication. The NOS interface described in this report represents the lowest sublayer in the OSI applications layer and may actually only be seen by the applications programmer who is designing and implementing these agent processes. Indeed, processes at the subsystem level may not even be aware the NOS is used for some purpose. For example, a DMS user may wish to view information in a database. The information may be in a database local to the network node or on a remote node. The process that handles the operator query may pass the query on to an information system that in turn must determine how to access the requested information. Whether the information system can locate the information locally or must request services from a remote information system may be entirely transparent to the process that handled the operator's query.

Section 3

REQUIREMENTS

Our services are designed to fulfill the following requirements:

1. Virtual-circuit (connection) service shall be provided. It shall be possible to associate a priority with such a connection to guarantee that the transport lag will not exceed a specified upper bound. No communication shall take place over a virtual circuit until a connection has been established. Unidirectional virtual-circuit connections shall be provided. These can be combined to establish two-way communication.

2. Datagram (connectionless) service shall be provided. This will allow a message to be sent to another processing entity without first establishing a connection. A processing entity may have zero or more logical datagram input streams, and a datagram is addressed to a particular stream.

3. It shall be possible to establish virtual-circuit connections

   - by prior agreement between processing entities, without an exchange of messages at the application-code level (OSI level 7); or

   - in response to a received message (by a datagram or over an existing virtual circuit), in accordance with a previously agreed-upon protocol, requesting that a virtual circuit be established for transmission of messages belonging to one of a previously agreed upon finite set of data types.

4. In establishing virtual circuits, it shall be possible for application programmers to specify certain properties of the connection. These properties might include the maximum acceptable transport lag, the degree of error checking required, security constraints, whether or not network resources supporting the connection should remain committed to the connection between transmissions, and the relative priority of different connections when contending for network resources.

5. In requesting virtual-circuit connections and addressing datagrams, only logical names shall be used. These logical names shall make no reference to the physical location or underlying implementation of a processing entity.

6. In requesting virtual-circuit connections and addressing datagrams, a processing entity outside of processing entity X shall not refer to the internal structure of X. In particular, it shall not refer to lower-level processing entities contained in X.

7. Once a virtual-circuit connection has been established between two processing entities, but not before then, the NOS shall provide connection end-point identifier values to each processing entity. All calls on the NOS to perform virtual-circuit communications shall identify with a connection end-point identifier the virtual circuit to be used.

8. A single virtual circuit or a single datagram stream shall be restricted to messages of a single type. It shall be possible for Ada compile-time consistency checks to detect attempts to use the same virtual-circuit connection for data of different types.

9. It should be possible for application programmers to isolate the decision to communicate using virtual circuits or datagrams, limiting the amount of program text that must be modified if this decision is changed.

10. Flexibility should be provided to interleave the receipt of data with other processing. In particular:

    - It must be possible to wait for incoming data without busy waiting.
    - It must be possible to perform other activities while waiting for data to arrive.
    - It must be possible to limit the amount of time a process waits for data to arrive.
    - There should be a straightforward way to process several incoming streams of data whose arrivals are interleaved.

Section 4

SERVICES

This section describes the application programmer's interface to the network operating system. The section consists of four subsections:

1. An overview explaining the approach taken and the basic elements of the interface. This subsection is tutorial, concentrating on concepts and omitting details.
2. The actual syntactic interface, in the form of an Ada generic package specification.
3. A detailed specification of the behavior of each subprogram provided as part of the interface, including the exceptions that may be raised by each subprogram.
4. A cross-reference by exception of the circumstances in which each exception may be raised by the various subprograms.

4.1 Overview

In designing our services, we have followed the design principle of parsimony. Rather than introducing new conceptual models, we have built upon the Ada conceptual model for file input/output to model network communication. We have tried to provide the simplest interface that will allow an application to obtain the required effects in a straightforward way. When certain requirements can be met by use of Ada tasking constructs, we have not duplicated those constructs in the network communication services.

An Ada program generally uses input/output operations to send data to or receive data from entities outside the program. These entities, known as external files, have traditionally been files in a file system (for example, disk files) or devices (for example, a keyboard, a video display, or a printer). An Ada program uses the Network Operating System to send data to or

receive data from entities outside the program, so it is appropriate to model network communications as input/output operations. In this case, external files correspond to virtual circuit connections or datagram streams.

Network communications services are provided to the Ada programmer through a generic package called Network_IO. This generic package closely resembles the predefined generic package Sequential_IO, the primary differences being in the treatment of external files. Like Sequential_IO, Network_IO is instantiated once for each type of data to be written or read. Any programmer familiar with predefined Ada input/output will find Network_IO easy and comfortable to work with. Sequential_IO has a simple interface, and Network_IO inherits this simplicity.

Processing entities may communicate through virtual circuits or by sending datagrams. Some internal files (that is, file variables in a program) are used for virtual-circuit communication and some for datagram communication. For virtual circuit communication, an open file is (in the jargon of the OSI reference model) a connection end point. For datagram communication, an internal output file corresponds to a stream of datagrams addressed to a particular processing entity; and an internal input file corresponds to a stream of datagrams addressed to the processing entity executing the program. Sections 4.1.1 and 4.1.2 describe scenarios for virtual-circuit and datagram communication, respectively, in greater detail.

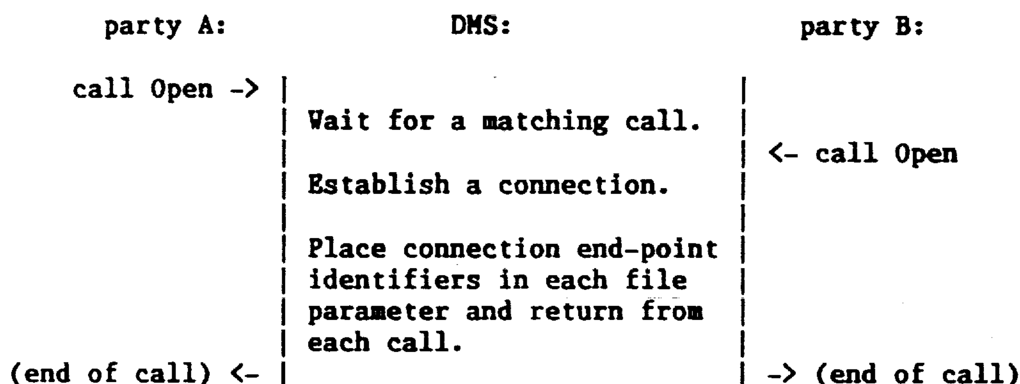


Figure 4-1. Calls on Open Waiting Until Both Parties Have Requested a Connection and the Connection Has Been Established

There is a potential requirement for limited multicasting or broadcasting of messages in emergency situations. The version of Network_IO presented in this document does not address this requirement because there are many open issues that must be resolved before the requirements are crystallized. Section 4.1.3 discusses these issues.

The use of Network_IO does not in any way require the use of multitasking. This is an important practical concern because of the inefficiency of currently available Ada tasking implementations. Section 4.1.4 addresses these concerns in greater detail.

4.1.1 Virtual-Circuit Communication

A virtual circuit is established by opening a file and disconnected by closing the file. Connections are unidirectional, with the direction of data transmission determined by the Mode parameter to Open. Unlike Sequential_IO, which uses a string to pass the name of an external file to Open, Network_IO uses a string to pass the logical name of a processing entity to Open. The Form parameter of Open is a string that describes properties of the connection and of the negotiation for a connection. For a connection to be established, both parties to the communication must call Open. This may happen by prior agreement between the parties or in response to one party sending a datagram to the other, for example. As shown in Figure 4-1, neither call on Open completes until the NOS has received a call from both parties and established a connection. When the calls complete, each internal file variable identifies an end point of the connection. Each party may view the effect of a normal call on Open as establishing a connection, without regard to the actions of the other party.

A call on Write transmits data over the connection and a call on Read obtains transmitted data. Transmitted messages are buffered by the NOS, so additional calls on Write can take place before a call on Read obtains the data transmitted by an earlier call on Write. Normally, if no data is available for receipt, a call on Read waits for such data to become available.

However, a procedure Set_Data_Unavailable_Response can be called to change the native of a given file so that a Read operation invoked when no data is available will raise the exception Data_Unavailable immediately. When a file's data-unavailable response is to fail, an unblocked input operation can be achieved by a block statement containing a call on Read for that file and a handler for Data_Unavailable.

When the transmitting processing entity has written its last message, it calls Close to sever the connection. Any data previously sent but not yet read remains buffered so that it can be read by the receiving processing entity. Once this data has been read, any attempt to read more data will raise the exception End_Error (the exception raised by the standard Ada input/output packages when an attempt is made to read past the end of a file). The function End_Of_File can be applied to an input file to avoid raising this exception. The function returns False if there is data in the corresponding buffer. It returns True if the buffer is empty and the sending processing entity has called Close. Otherwise, the call waits for the sending processing entity to either place another message in the buffer or to call Close. The resulting relationship between a writer's calls on Write and Close and a reader's calls on End_Of_File and Read are identical to the relationship created by Sequential_IO. As observed by a single sending or receiving task, the behavior of these subprograms when used for communication is identical to the behavior of the corresponding Sequential_IO subprograms when used to write or read disk files.

The receiving processing entity may sever the connection at any time by calling Close, though it will typically do so only after End_Of_File becomes true. This has no effect on the sending processing entity. It is analogous to one program writing one hundred records to a disk file and another program reading the first ten, then closing its input file. Any data buffered in a virtual-circuit connection when the receiving processing entity calls Close is permanently lost.

4.1.2 Datagram Communication

Datagrams are addressed not to processing entities, but to datagram streams. There are zero or more datagram streams associated with a processing entity. These streams have names assigned by the network administrator and exist permanently unless removed by the network administrator. Datagram stream names for a given processing entity are unique, but different processing entities may have identically named streams, so a datagram stream is identified by the combination of a processing-entity name and a stream name.

A processing entity may have more than one datagram stream. A given datagram stream is used for transmitting messages of only one data type. By prior agreement, different streams of the same data type might be used for transmitting messages of different priorities. A processing entity can then process highest-priority datagrams first by checking streams for input in priority order. This approach is illustrated in Section 5.6.

Given the name of a datagram stream (consisting of a processing entity name plus a stream name), the function Datagram_Output_File returns an internal-file value corresponding to that stream. Given the name of one of the executing processing entity's own datagram streams, the function Datagram_Input_File returns an internal-file variable corresponding to that incoming datagram stream. Because the type for internal files is limited private and no provision is made for copying datagram-stream internal files into variables, internal files corresponding to datagram streams can only be named by such function calls.

Conceptually, the internal files associated with datagram streams are always open, even when the corresponding processing entities are inactive. Indeed, since these internal files cannot be named by variables, they cannot be passed to the Open or Close procedures, whose internal-file parameters are of mode in out. (Datagram-stream files are analogous in this respect to the standard input and output files of the predefined Ada package Text_IO.) Datagram-stream files may be passed to the End_Of_File function, but that function always returns False for such files.

A datagram is written to a particular stream of a particular processing entity by a call of the form

```
Write (Datagram_Output_File (Entity, Stream), Message);
```

A processing entity reads a datagram from a particular incoming stream by a call of the form

```
Read (Datagram_Input_File (Stream), Message);
```

From the caller's point of view, the behavior of Write and Read for datagram-stream internal files is identical to their behavior for virtual-circuit internal files. The only difference is how those internal files are generated.

If a datagram stream is empty and the corresponding file's data-unavailable response is to wait, a task calling Read is blocked until a datagram arrives. However, other tasks in the same processing entity may continue to execute. If the task performing a Read accepts an entry call just afterward to pass the message on to another task, a conditional call on this the entry has the effect of obtaining an incoming datagram if one is available and performing a specified alternative action otherwise. Like virtual-circuit internal files, datagram internal files can be set to trigger the Data_Unavailable exception if Read is called when no data is available. An application requiring incoming datagrams to be fetched in a certain order, according to some property of the message contents, can dedicate a task to fetching datagrams from an incoming stream as quickly as they arrive and inserting them in a data structure based on the relevant properties. The same task would accept entry calls to extract datagrams with specified properties from the data structure. This approach is illustrated in Section 5.7.

4.1.3 Broadcast and Multicast Services

There may be a requirement to send the same message to several recipients simultaneously. This is known as **multicasting**. A special case of multicasting is **broadcasting**, sending the same message to all possible recipients. We are aware of two roles a multicast capability can play. The first is to propagate alarm messages in the event of a critical emergency. The second is to provide a way for an application to send messages to a class of recipients (e.g., to all engineering

consoles or to all subsystems requiring a time stamp) without requiring that the sending program be modified every time a processing entity is added to or removed from that class.

Network_IO does not explicitly provide multicast capability, but there are several ways to provide such a capability within the Network_IO model. Several open issues must be resolved before a multicast capability can be specified. Resolution of these issues, in turn, requires clarification of the requirements for a multicast capability. This section outlines the relevant issues and sketches several alternative approaches.

Multicast messages must be datagrams. A virtual-circuit connection is, by its nature, a relationship between two parties that has been explicitly negotiated by both parties. Therefore, multicasting must consist of sending multiple copies of a single datagram to different datagram streams.

It is an underlying principle of Network_IO that each datagram stream is restricted to messages of a single type. This is practically required by the strong type system of the Ada language: A subprogram used to obtain incoming data can only obtain data of one type; distinct subprograms are required to obtain data of different types.*

Thus it does not make sense to broadcast a message to all datagram streams. Messages can only be sent to streams of the appropriate type. This suggests that, if broadcasting is to be supported, all broadcast datagrams should be of the same type. The broadcast cannot be to all datagram streams, but only to streams of this type.

---

*If subprograms are overloaded, there are still distinct subprograms, even though they happen to have the same name. There must be enough information at the point of the subprogram call (for example, the types of the actual parameters) for the compiler to determine which one of the distinct subprograms with a common name is being called. Generic units are applicable to multiple types, but only after they are instantiated. A program does not call a generic subprogram, but a specific instance of a generic subprogram.

Ada's type restrictions support a fundamental design principle: A processing entity should not be sent a message unless it is expecting that kind of message and is prepared to handle it. Unlike people, processing entities cannot be expected to receive information of an unanticipated form and invent an appropriate response. If a processing entity is to receive an emergency datagram, it should be via a stream dedicated to that purpose.

It is not clear that a universal broadcast is appropriate in an emergency. Certain emergencies will have to be announced to many subsystems throughout the network, but there may be many processing entities that have no sensible response to such an emergency. Broadcasting announcements to all processing entities complicates programming, because each processing entity must recognize the emergency announcements, even if only to ignore them. More seriously, unnecessary broadcasting may slow down the transmission of the emergency announcements to the processing entities where they are really needed and divert system resources needed to respond to the emergency.

The role of each subsystem in responding to an emergency must be carefully planned by system integrators. Universal broadcast can lead to the dangerous assumption that, since emergency announcements have been broadcast to each of several independently developed processing entities, the problem is somehow taken care of. In fact, if a processing entity has an important role to play in an emergency situation, this should be part of its interface specification. In that case, a conscious decision will be made to place the processing entity on a multicast list for emergency announcements.

The current model of Network_IO is capable of supporting multicasting. One naive approach is simply for processing entities that must multicast to explicitly transmit a separate copy of the datagram to all intended recipients:

```
for I in Multicast_List'Range loop
   Write
      (Datagram_Output_File (Multicast_List (I), "ALARM_STREAM"),
       Message);
   end loop;
-- Multicast_List is an array of strings.
```

There are several problems with this approach:

- The list of recipients must be managed by the program, requiring the program (or a file read by the program) to be changed every time a recipient is added or removed. It might be preferable for the network administrator to maintain a single list that is somehow identified by the application program.
- Datagram_Output_File must be called anew for each recipient. It is impossible to compute internal-file values and store them in a table ahead of time, because these values belong to a limited-private type.
- Given the knowledge that the same message is to go to several recipients, the NOS may be able to deliver datagrams more efficiently than by repeated transmission of individual copies.

The first of these problems can be solved within the current design of Network_IO by establishing "a post-office processing entity" for each class of recipients. The post-office processing entity would simply relay any message it received to each processing entity on its recipient list. There would be only one copy of the recipient list, maintained by the system administrator.

The second and third problems could be solved by implementing the post-office processing entities as extensions of the NOS rather than as ordinary application processing entities. Internally, the post-office processing entities would be implemented in terms of low-level NOS operations rather than in terms of Network_IO, allowing the knowledge that there are multiple recipients to exploited efficiently; but they would appear from the outside to be ordinary processing entities. No change would be necessary to the Network_IO interface provided to applications.

(Equivalently, the post-office processing entities could be thought of as <u>virtual processing entities</u> that exist in name only. The NOS intercepts all datagrams addressed to such processing entities and handles them specially, by multicasting them. From the application programmer's point of view, addressing a datagram to such a processing entity has the same effect as addressing it to a multicast list.)

4.1.4 The Role of Tasking

It is both convenient and stylistically appropriate to use Network_IO in conjunction with multitasking, but this is by no means required. The specification of Network_IO does not include tasks. Furthermore, an application may use Network_IO without using multiple Ada tasks. Therefore, concerns about the performance of multitasking in early Ada implementations do not impede the effective use of Network_IO.

While Network_IO does not require the use of multitasking, it can be used to its fullest potential in multitask designs. Order-of-magnitude improvements in the performance of Ada multitasking implementations can be expected in the near future and throughout the life of the Space Station. Once these improvements are realized, important benefits will be achieved by transition to a multitask approach. The design of Network_IO will facilitate this transition.

For an application receiving data from multiple asynchronous sources or performing other processing while waiting for messages to arrive, the use of multitasking greatly simplifies the logic of the application, resulting in lower development costs, higher reliability, and substantially safer and easier modification of the program. Concurrency is natural in such an application, because there are several separate conceptual threads of events with which the program must deal. By associating a task with each such thread, one can construct a program whose structure corresponds directly to the problem to be solved. Typically, this means providing one task to handle each asynchronous source of input plus a central task to perform the main processing. Ada's tasking features provide a straightforward way for the programmer to control the synchronization of these tasks, simply and explicitly, while isolating the logic of independent conceptual threads.

As a practical matter, most early implementations of rendezvous are too slow to be used for handling high-volume communications. To cope with this reality, Network_IO provides the ability to specify that a call on Read will raise the exception Data_Unavailable if no data is waiting to be read. This

allows a single-task program (i.e., a program without any task units) to poll for data and to interleave other processing with the polling. The interleaving logic will be far more complex than the logic of a multitask program, but the nonmultitasking use of Network_IO will result in no more complexity than any other nonmultitasking approach.

4.2 Specification of Network_IO

Below is the Ada specification of the generic package Network_IO. The specification lists the types, subprograms, and exceptions constituting the application programmer's interface to the Network Operating System and describes the syntax for invoking the subprograms. The behavior of the subprograms is described in Section 3.

```
with IO_Exceptions;

generic

   type Message_Type is private;

package Network_IO is

   type File_Type is limited private;
   type File_Mode is (In_File, Out_File);
   type Data_Unavailable_Response_Type is (Wait, Fail);

   procedure Open
      (File : in out File_Type;
      Mode : in File_Mode;
      Name : in String;
      Form : in String);

   procedure Close (File : in out File_Type);

   function Datagram_Output_File
      (Entity_Name : String;
       Stream_Name : String)
      return File_Type;

   function Datagram_Input_File
      (Stream_Name : String) return File_Type;
```

```
   procedure Set_Data_Unavailable_Response
      (File     : in File_Type;
       Response : Data_Unavailable_Response_Type);

   procedure Read (File : in File_Type; Item : out Message_Type);

   procedure Write (File : in File_Type; Item : out Message_Type);

   function End_Of_File (File : in File_Type) return Boolean;

   Data_Unavailable : exception;
   Status_Error     : exception renames IO_Exceptions.Status_Error;
   Mode_Error       : exception renames IO_Exceptions.Mode_Error;
   Name_Error       : exception renames IO_Exceptions.Name_Error;
   Use_Error        : exception renames IO_Exceptions.Use_Error;
   Device_Error     : exception renames IO_Exceptions.Device_Error;
   End_Error        : exception renames IO_Exceptions.End_Error;
   Data_Error       : exception renames IO_Exceptions.Data_Error;

private

   type File_Type is [irrelevant to user of Network_IO];

end Network_IO;
```

4.3 Behavior of Network_IO Subprograms

This section describes the behavior of the seven subprograms provided by an instance of the generic package Network_IO. In each case we begin by repeating the Ada subprogram specification given in the generic package specification of Section 2. This is followed by English text describing the behavior of the subprogram. Finally, we enumerate the exceptions that may be raised by each subprogram and the circumstances in which those exceptions are raised.

4.3.1 Open

```
procedure Open
   (File : in out File_Type;
    Mode : in File_Mode;
    Name : in String;
    Form : in String);
```

A call on Open establishes a virtual-circuit connection to the processing entity named by the Name parameter. The establishment of the connection obeys the constraints specified in the Form parameter. The File parameter is set to an open internal-file value identifying an end point of this connection. If

the Mode value specified in the call on Open is Out_File, messages may be sent over this connection by writing to the internal file. If the Mode value specified in the call on Open is In_File, messages may be received over this connection, in the order in which they were sent, by reading from the internal file.

[The syntax of the Form string and the full range of attributes specifiable by the string are still to be determined. We have identified the following attributes so far:

- **the <u>role</u> of the requested connection, allowing the NOS to correctly match multiple connection requests by the same pair of processing entities**
- **the quality of service, including:**
  - **the maximum acceptable transport lag for data transmission**
  - **whether bandwidth is to be reserved during the lifetime of the connection or seized and released with each transmission**
  - **whether error checking and correction is required**
- **the time limit within which the other party must also request the connection before the request is abandoned**
- **security-related constraints**

Using a string instead of a record type requires run-time interpretation of the string. However, it allows new Form attributes to be defined during the evolution of the NOS without recompiling all code that calls Open. Furthermore, it is consistent with the Form parameter in Ada's predefined input/output packages.]

<u>Exceptions</u>:

- Open raises Status_Error if the internal file passed as File is already open.
- Open raises Name_Error if the string passed as Name is not a valid logical name of a processing entity.
- Open raises Use_Error if a virtual connection cannot be established subject to the constraints specified in the Mode parameter (including the constraint to establish a connection within a certain amount of time).
- Open raises Device_Error if a malfunction or overloading of the NOS prevents completion of a connection.

4.3.2 Close

```
procedure Close (File : in out File_Type);
```

Close severs the association between the internal file passed as File and the corresponding connection end point, allowing any NOS resources used by that association to be released. Despite the severing of the association between the internal file and the connection end point, the connection itself remains in existence, preserving data that has been sent but not yet received, until neither end point is associated with an internal file.

(**Note:** Programs opening files should close them once the files are no longer needed, so that resources can be released.)

Exceptions:

- Close raises Status_Error if the internal file passed as File is not open.

4.3.3 Datagram_Output_File

```
function Datagram_Output_File
   (Entity_Name : String;
    Stream_Name : String)
   return File_Type;
```

Datagram_Output_File returns an open internal file corresponding to the datagram stream named by Stream_Name belonging to the processing entity named by Entity_Name. A datagram may be sent to that stream by writing to that internal file.

Exceptions:

- Datagram_Output_File raises Name_Error if the string passed as Entity_Name is not the valid logical name of a processing entity, or if it is a valid name but Stream_Name does not name one of that processing entity's datagram streams.

- Datagram_Output_File raises Device_Error if a malfunction or overloading of the NOS prevents interpretation of the strings passed as Entity_Name and Stream_Name.

4.3.4 Datagram_Input_File

```
function Datagram_Input_File
   (Stream_Name : String) return File_Type;
```

Datagram_Input_File returns an open internal file corresponding to the datagram stream named by Stream_Name belonging to the processing entity executing the call. A datagram may be retrieved from that stream by reading from that internal file.

Exceptions:

- Datagram_Input_File raises Name_Error if the string passed as Stream_Name does not name a datagram stream of the processing entity executing the call.

4.3.5 Set_Data_Unavailable_Response

```
procedure Set_Data_Unavailable_Response
   (File     : in File_Type;
    Response : Data_Unavailable_Response_Type);
```

Sets the current data-unavailable response for the input file passed as File to the value passed as Response. A file's current data-unavailable response affects the behavior of Read. The setting remains in effect until termination of the main program or until another call on Set_Data_Unavailable_Response. The initial data-unavailable response of a file (set for virtual-circuit connection files when they are opened and for datagram-stream files at the start of the main program) is Wait.

Exceptions:

- Set_Data_Unavailable_Response raises Status_Error if the file passed as File is not open.
- Set_Data_Unavailable_Response raises Mode_Error if the file passed as File is open with a mode of Out_File.

4.3.6 Read

```
procedure Read (File : in File_Type; Item : out Message_Type);
```

Read receives the next message from the input file passed as File and places its value in the Item parameter. If the current data-unavailable response of the file passed as File is Fail, the exception Data_Unavailable is raised. Unless it raises an exception, a call on Read does not complete until a message is received.

Exceptions:

- Read raises Data_Unavailable if the current data-unavailable status of the file passed as File is Fail and no message is immediately available to be read.
- Read raises Status_Error if the internal file passed as File is not open.
- Read raises Mode_Error if the internal file passed as File is open with a mode of Out_File.
- Read raises Device_Error if a malfunction or overloading of the NOS prevents receipt of the next message.
- Read raises End_Error if the internal file passed as File is associated with a virtual-circuit connection, the sender's connection end point is no longer associated with an internal file, and the connection contains no unreceived data.
- In some cases, Read may raise Data_Error if the message read cannot be interpreted as a value of the type Message_Type. However, the NOS is not required to raise this exception in all cases.

4.3.7 Write

```
procedure Write (File : in File_Type; Item : out Message_Type);
```

Write sends the value passed as Item to the output file passed as File.

Exceptions:

- Write raises Status_Error if the internal file passed as File is not open.

- Write raises Mode_Error if the internal file passed as File is open with a mode of In_File.

- In some cases, Write may raise Device_Error if a malfunction or overloading of the NOS prevents receipt of the message. However, the NOS is not required to raise this exception in all cases.

  [For a virtual-circuit file, the raising of Device_Error will depend on the degree of error-checking specified in the Form parameter when the file was opened.]

4.3.8 End_Of_File

```
function End_Of_File (File : in File_Type) return Boolean;
```

Returns True if the internal file passed as File is associated with a virtual-circuit connection, the sender's connection end point is no longer associated with an internal file, and the connection contains no unreceived data. Returns False if the internal file passed as File is associated with a datagram stream or if the internal file passed as File is associated with a virtual-circuit connection that still contains unread data. If the internal file passed as File is associated with a virtual-circuit connection, the sender's connection end point is still associated with an internal file, and the connection contains no unread data, End_Of_File does not return a value until the sender either severs the association between the connection and its output file (in which case True is returned) or sends another message (in which case False is returned).

This behavior is summarized in Figure 4-2. Its net effect is to return True for virtual-circuit files over which more data can be read, to return False for virtual-circuit files over which there will be no more data to be read, and to return False for all datagram-stream files at all times.

<table>
<tr><th>case</th><th>kind of file</th><th>unread data left?</th><th>sender's file open?</th><th>action</th></tr>
<tr><td>1</td><td rowspan="4">virtual circuit</td><td rowspan="2">yes</td><td>yes</td><td rowspan="2">return False</td></tr>
<tr><td>2</td><td>no</td></tr>
<tr><td>3</td><td>no</td><td>yes</td><td>Wait for case 1 or 4 to hold.</td></tr>
<tr><td>4</td><td>no</td><td>no</td><td>return True</td></tr>
<tr><td>5</td><td rowspan="2">datagram</td><td>yes</td><td rowspan="2">N/A</td><td rowspan="2">return False</td></tr>
<tr><td>6</td><td>no</td></tr>
</table>

Figure 4-2. Behavior of the End_Of_File Function

Exceptions:

- End_Of_File raises Status_Error if the internal file passed as File is not open.
- End_Of_File raises Mode_Error if the internal file passed as File is open with a mode of Out_File.
- For virtual-circuit files, End_Of_File raises Device_Error if a malfunction or overloading of the NOS prevents checking whether the connection's other end point is still associated with an internal file or whether the connection contains unread data.

4.4 Summary of Exceptions

This section reviews the information given in Section 3 about the raising of exceptions. In this section, however, the information is organized by exception instead of by subprogram.

4.4.1 Data Unavailable

Read raises Data_Unavailable if the current data_unavailable status of the file passed as File is Fail and no message is immediately available to be read.

4.4.2 Status_Error

Open raises Status_Error if the internal file passed as File is already open. Close, Set_Data_Unavailable_Response, Read, Write, and End_Of_File raise Status_Error if the internal file passed as File is not open.

4.4.3 Mode_Error

Set_Data_Unavailable_Response, Read and End_Of_File raise Mode_Error if the internal file passed as File is open with a mode of Out_File. Write raises Mode_Error if the internal file passed as File is open with a mode of In_File.

4.4.4 Name_Error

Open raises Name_Error if the string passed as Name is not a valid logical name of a processing entity. Datagram_Output_File raises Name_Error if the string passed as Entity_Name is not the valid logical name of a processing entity, or if it is a valid name but Stream_Name does not name one of that processing entity's datagram streams. Datagram_Input_File raises Name_Error if the string passed as Stream_Name does not name a datagram stream of the processing entity executing the call.

4.4.5 Use_Error

Open raises Use_Error if a virtual connection cannot be established subject to the constraints specified in the Mode parameter (including the constraint to establish a connection within a certain amount of time).

4.4.6 Device Error

In general, an operation raises Device_Error if a malfunction or overloading of the NOS prevents completion of the operation. Open raises Device_Error if a connection cannot be established. Datagram_Output_File raises Device_Error if the strings passed as Entity_Name and Stream_Name cannot be interpreted. Read raises Device_Error if the next message cannot be received. In some cases (depending on the level of error checking specified when a virtual-circuit connection is established), Write may raise Device_Error if the message being written cannot be received. For virtual-circuit files, End_Of_File raises Device_Error if checks cannot be made to determine whether the connection's other end point is still associated with an internal file or whether the connection contains unread data.

4.4.7 End Error

Read raises End_Error if the internal file passed as File is associated with a virtual-circuit connection, the sender's connection end point is no longer associated with an internal file, and the connection contains no unreceived data.

4.4.8 Data Error

In some cases, Read may raise Data_Error if the message read cannot be interpreted as a value of the type Message_Type. However, the NOS is not required to raise this exception in all cases.

Section 5

EXAMPLES OF USE

This section contains eight hypothetical applications illustrating how Network_IO may be used to achieve a wide range of flexible network-communication capabilities. Each example contains actual Ada code, though in some cases we have used bracketed, underlined text in place of code dealing with details irrelevant to network communications. We have used two kinds of illustrations to represent the structure of the examples graphically.

To represent the pattern of network communications, we use diagrams like Figure 5-1. Circles represent processing entities and rectangular strips extending from the circles represent datagram streams. A jagged arrow from a circle to a strip representing a datagram stream indicates that a datagram may be sent from the processing entity represented by the first circle to the datagram stream represented by the strip. A virtual-circuit connection between two processing entities is represented by a line with a plug at each end plugged into the corresponding circle. An arrowhead on the line indicates the direction of the connection. The diagram depicts **potential** communication paths. It does not indicate the circumstances under which datagrams are actually sent or virtual circuits are actually established, nor the order in which these things happen.

To represent the structure of an Ada program, we use diagrams like Figure 5-2. Large parallelograms represent concurrent tasks and small embedded parallelograms represent entries of that task. Rectangles with embedded rectangles represent packages and the embedded rectangles represent subprograms provided by that package. Stand-alone rectangles represent separately compiled subprograms. Large solid arrows represent procedure and entry calls and point from the caller to the called subprogram or entry.

Small arrows represent data passed through parameters during a subprogram or entry call and point from the producer of the data to its consumer.

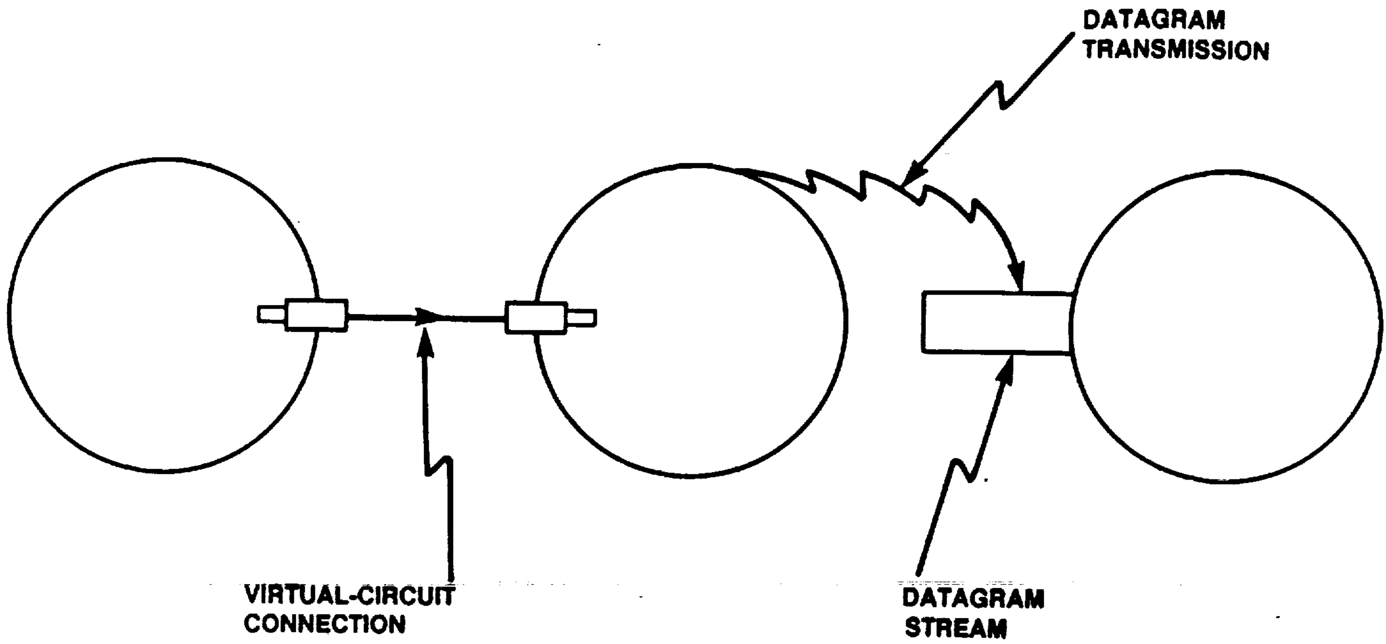


Figure 5-1. Notation for Network Communication Patterns

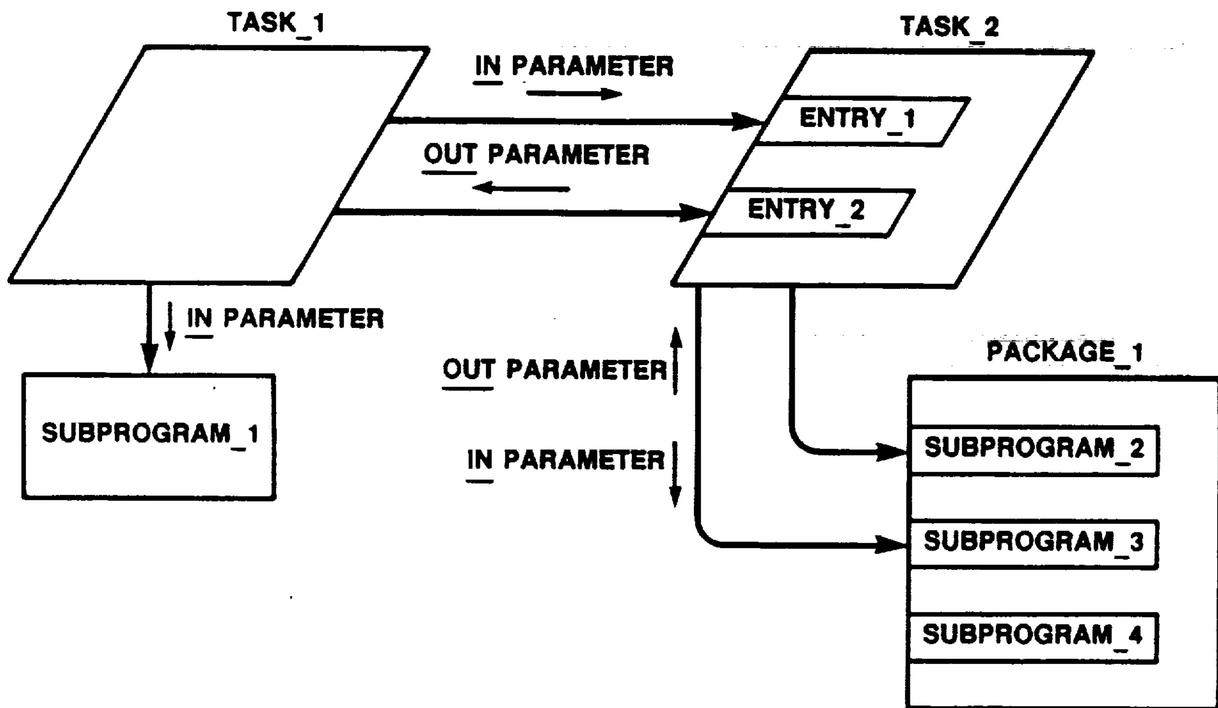


Figure 5-2. Notation for Program Structure

5.1 Processing a Bounded Sequence of Data

By prior agreement, the processing entity DATA_ANALYZER and the processing entity PAYLOAD_DATA_MANAGER are to establish a virtual-circuit connection. PAYLOAD_DATA_MANAGER will transmit a sequence of Float values to DATA_ANALYZER, then the connection will be removed. Figure 5-3 depicts the flow of messages across the network.

Both processing entities instantiate Network_IO as follows:

```
package Network_Float_IO is new Network_IO (Float);
```

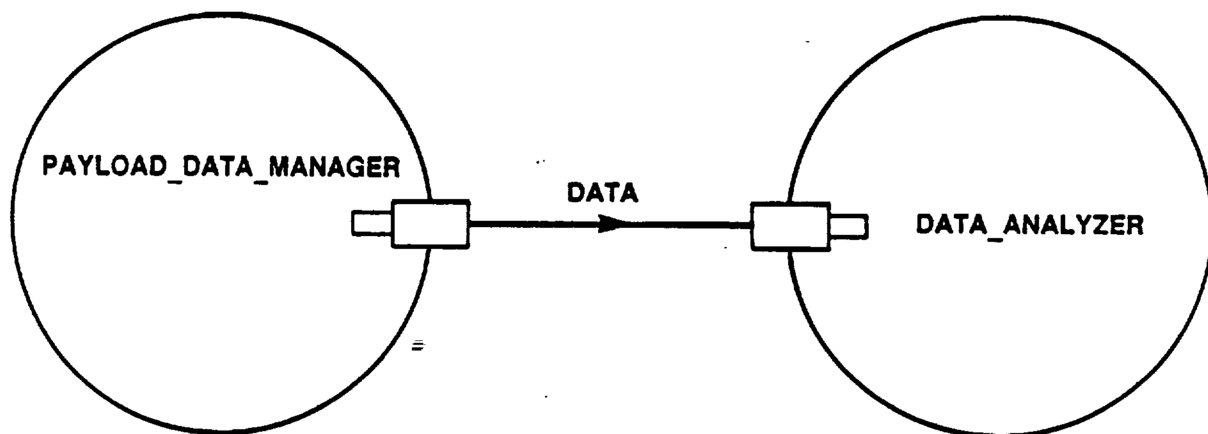


Figure 5-3. Flow of messages from PAYLOAD_DATA_MANAGER to DATA_ANALYZER

PAYLOAD_DATA_MANAGER executes statements of the following form, where Output_File is a variable of type Network_Float_IO.File_Type, Form is a variable of type String, and X is a variable of type Float:

```
Network_Float_IO.Open
   (Output_File,
    Network_Float_IO.Out_File,
    "DATA_ANALYZER",
    Form);

while [there is additional data to send] loop
   [obtain the data and place it in X];
   Network_Float_IO.Write (Output_File, X);
end loop;

Network_Float_IO.Close (Output_File);
```

DATA_ANALYZER executes statements of the following form, where Input_File is a variable of type Network_Float_IO.File_Type, Form is a variable of type String, and Y is a variable of type Float:

```
Network_Float_IO.Open
   (Input_File,
    Network_Float_IO.In_File,
    "PAYLOAD_DATA_MANAGER",
    Form);

while not Network_Float_IO.End_Of_File (Input_File) loop
   Network_Float_IO.Read (Input_File, Y);
   [process the data in Y];
end loop;

Network_Float_IO.Close (Input_File);
```

5.2 Performing Background Processing While Waiting for Datagrams

A processing entity is responsible for taking sensor measurements every second and maintaining the average of the five most recent measurements. Every time a datagram arrives on the incoming stream REQUEST_STREAM, a return datagram specifying the most recent average is to be sent to the stream named in the incoming datagram. (The incoming datagram is analogous to ordinary

mail containing a return-address label to be used for sending a reply.) The processing entity is to continue executing indefinitely. Figure 5-4 depicts the flow of messages across the network.

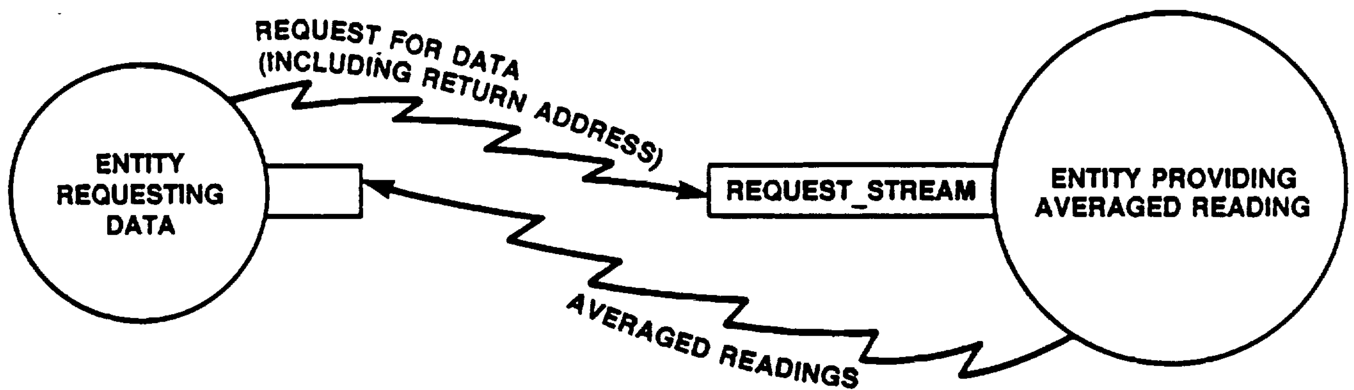


Figure 5-4. Flow of Requests for Data and Averaged Sensor Readings

The solution for the processing entity providing averaged readings consists of four compilation units:

- The package Request_Package. This package is imported (using a <u>with</u> clause) by the processing entity that computes averages and all processing entities that communicate with that processing entity, to define the common data type these processing entities use for requests for data.

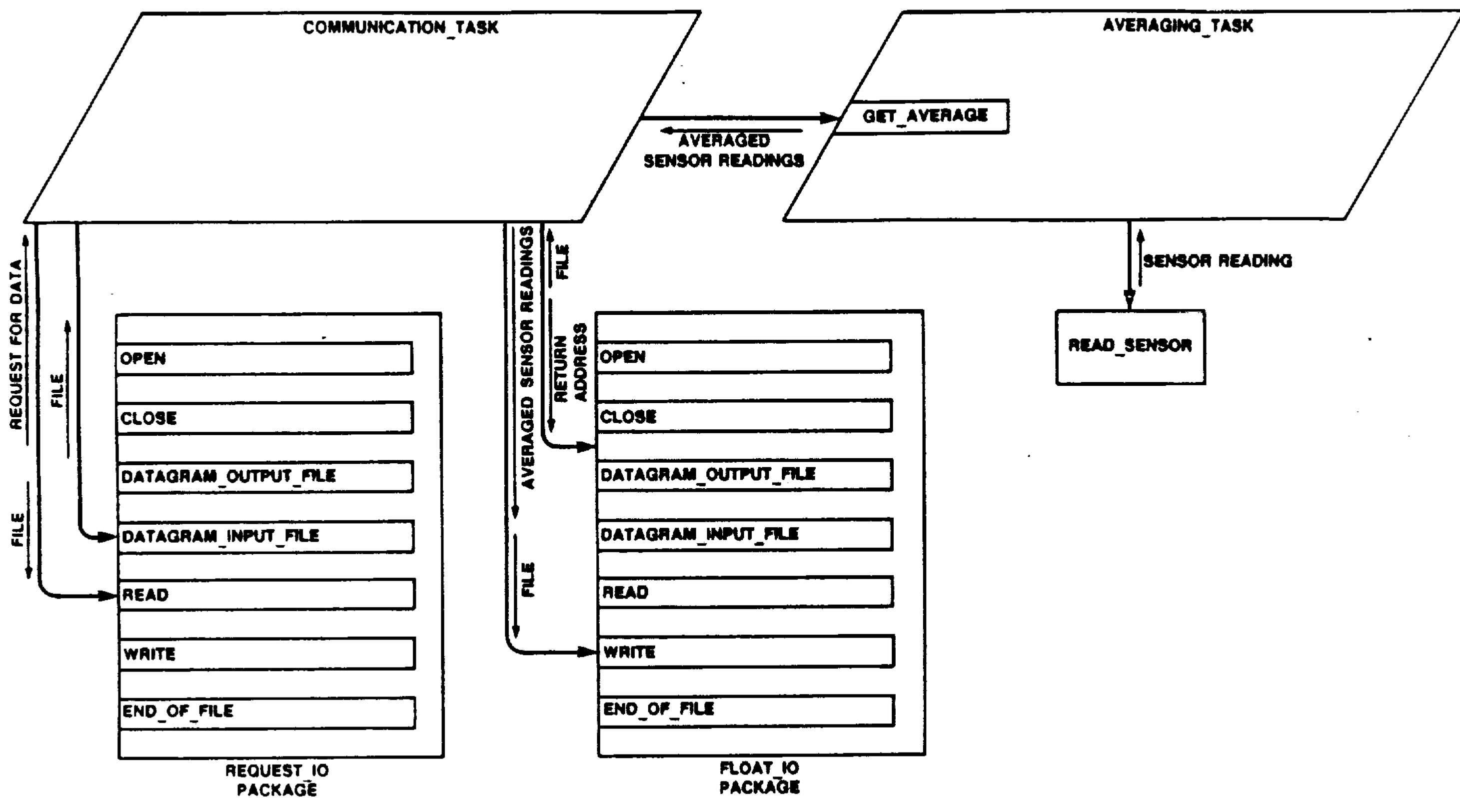
- A main program consisting of a task Communication_Task that receives and replies to datagrams and a task Averaging_Task that takes periodic measurements and provides averages upon request.

- A subunit for the Communication_Task body, showing how Network_IO is used.

- A subunit for the Averaging_Task body.

The Averaging_Task body is not directly relevant to network communications, but is supplied for completeness. We do not show the library procedure Read_Sensor imported by Averaging_Task to actually drive the sensor and obtain a reading. Figure 5-5 shows the relationships of the program units and the data flow among them.

```
package Request_Package is

   type Request_Type is
      record
         Requestor_Part : String (1 .. 128);
         Stream_Part    : String (1 .. 64);
      end record;

end Request_Package;
```



REQUEST_IO and FLOAT_IO are both instances of the generic package NETWORK_IO. COMMUNICATION_TASK executes a loop that is repeated once every time a request for sensor data is received. AVERAGING_TASK executes a loop that is executed once each time its entry is called and once each time it is time to read new data from the sensor. Neither task consumes processor cycles between repetitions of its loop.

Figure 5-5. Structure of the Program to Provide Average Readings on Request

```
procedure Main_Program is

   task Communication_Task;

   task Averaging_Task is
      entry Get_Average (Average : out Float);
   end Averaging_Task;

   task body Communication_Task is separate;
   task body Averaging_Task is separate;

begin

   null;  -- All work done by Communication_Task and
          -- Averaging_Task.

end Main_Program;


with Request_Package, Network_IO;

separate (Main_Program)

task body Communication_Task Is

   package Request_IO is
      new Network_IO (Request_Package.Request_Type);

   package Float_IO is new Network_IO (Float);

   Request : Request_Package.Request_Type;
   Average : Float;

begin

   loop

      Request_IO.Read
         (Request_IO.Datagram_Input_File ("REQUEST_STREAM"),
          Request);

      Averaging_Task.Get_Average (Average);

      Float_IO.Write
         (Float_IO.Datagram_Output_File
            (Request.Requestor_Part, Request.Stream_Part),
          Average);

   end loop;

end Communication_Task;
```

```
with Calendar, Read_Sensor; use Calendar;

separate (Main_Program)

task body Averaging_Task is

   Next_Reading_Time : Time;

   [other declarations]

begin

   -- Initialization.  No entry calls accepted until first five
   --    readings have been obtained.

   Next_Reading_Time := Clock;
   for I in 1 .. 5 loop
      [read the sensor and save the value read];
      Next_Reading_Time := Next_Reading_Time + 1.0;
      delay Next_Reading_Time - Clock;
   end loop;

   Next_Reading_Time := Next_Reading_Time + 1.0;

   -- Routine processing.  Wait for an entry call or
   --    Next_Reading_Time to arrive.

   loop
      select
         accept Get_Average (Average : out Float) do
            [place the average of the last five readings
             in Average];
         end Get_Average;
      or
         delay Next_Reading_Time - Clock;
         [read the sensor and update the list of readings];
         Next_Reading_Time := Next_Reading_Time + 1.0;
      end select;
   end loop;

end Averaging_Task;
```

Communication_Task remains blocked at the call on Read when waiting for an incoming datagram, without consuming CPU time. Similarly, Averaging_Task remains suspended at the selective wait until except when it is briefly woken up to retrieve a new reading or to deliver an average. It does not consume CPU time while suspended.

5.3 A File Server

A processing entity named FILE_SERVER has its own local file system. Other processing entities requiring data from this file system send a datagram to FILE_SERVER's datagram stream REQUEST_STREAM. FILE_SERVER responds by sending an acknowledgment datagram to a stream named in the request and then sending the contents of the file over a virtual circuit. (See Figure 5-6.) The records in the file are of type Float. The datagram sent to REQUEST_STREAM belongs to the type declared in the following package:

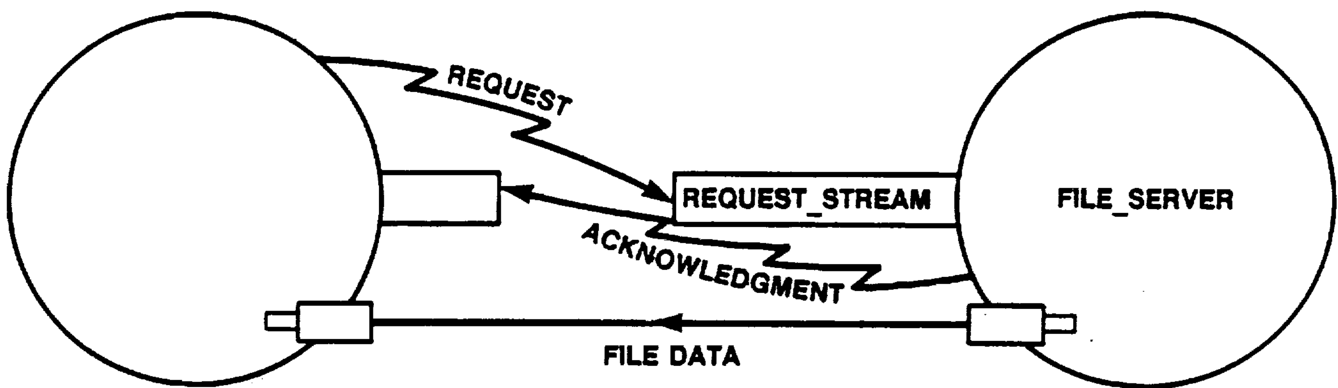


Figure 5-6. Flow of Messages Between FILE_SERVER and Client

```
package Service_Request_Package is

type Service_Request_Type is
   record
      Requestor_Name              : String (1 .. 64);
      Acknowledgment_Stream_Name : String (1 .. 64);
      File_Name                   : String (1 .. 64);
   end record;

end Service_Request_Package;
```

FILE_SERVER sends the acknowledgment datagram after trying to open the specified file. The acknowledgment is of type Boolean, equal to True if the file can be found and False otherwise. A virtual circuit connection is established only if the file can be found.

Here is the FILE_SERVER program:

```
with Service_Request_Package, Network_IO, Sequential_IO;

procedure File_Server is

   package Request_IO is
      new Network_IO
         (Service_Request_Package.Service_Request_Type);
   package Local_Float_IO is new Sequential_IO (Float);
   package Boolean_IO is new Network_IO (Boolean);
   package Remote_Float_IO is new Network_IO (Float);
   use Request_IO, Local_Float_IO, Boolean_IO, Remote_Float_IO;

   Request     : Service_Request_Package.Service_Request_Type;
   Input_File  : Local_Float_IO.File_Type;
   Successful  : Boolean;
   Output_File : Remote_Float_IO.File_Type;
   Data        : Float;

   Connection_Form : constant String := [some string literal];

begin

   loop

      Read (Datagram_Input_File ("REQUEST_STREAM"), Request);

      begin
         Open (Input_File, In_File, Request.File_Name);
         Successful := True;
      exception
         when Name_Error | Use_Error =>
            Successful := False;
      end;

      Write
         (Datagram_Output_File
            (Request.Requestor_Name,
             Request.Acknowledgment_Stream_Name),
         Successful);
```

```
    if Successful then

       Open
          (Output_File,
           Out_File,
           Request.Requestor_Name,
           Connection_Form);

       while not End_Of_File (Input_File) loop
          Read (Input_File, Data);        -- Local_Float_IO.Read
          Write (Output_File, Data);      -- Remote_Float_IO.Write
       end loop;

       Close (Output_File);

       Close (Input_File);

    end if;

 end loop;

end File_Server;
```

5.4 Converting Arriving Messages to Entry Calls

Often, incoming messages may arrive from several sources, interleaved in an unpredictable way. From an abstract point of view, this situation manifests several conceptual threads of activity. The most natural way to model this in the Ada language is with tasking.

Ada provides a rich set of facilities for handling interleaved stimuli from different sources. If such stimuli are presented to a central task as entry calls, then the selective wait statement provides the central task with a wide range of capabilities, including the following:

- to wait for stimuli from several sources and to respond to each stimulus as it arrives, based on the source of the stimulus
- to queue stimuli from a given source and only respond to them when certain conditions hold

- to perform other processing when no stimuli are waiting to be processed
- to perform other processing if no stimuli arrive within a specified amount of time

Such stimuli may include datagrams and virtual-circuit messages arriving from other processing entities, giving a processing entity great flexibility in the handling of incoming messages.

The following generic package can be instantiated for each desired virtual-circuit connection to produce a task that waits for input on that virtual circuit and calls a specified entry of a central task whenever a message arrives. A second specified entry is called upon successful or unsuccessful conclusion of the attempt to establish a connection and a third is called when the end of the file is detected. The type of the message, the name of the sending processing entity, the Form string to be used in establishing a connection, and the entries to be called are specified as generic parameters:

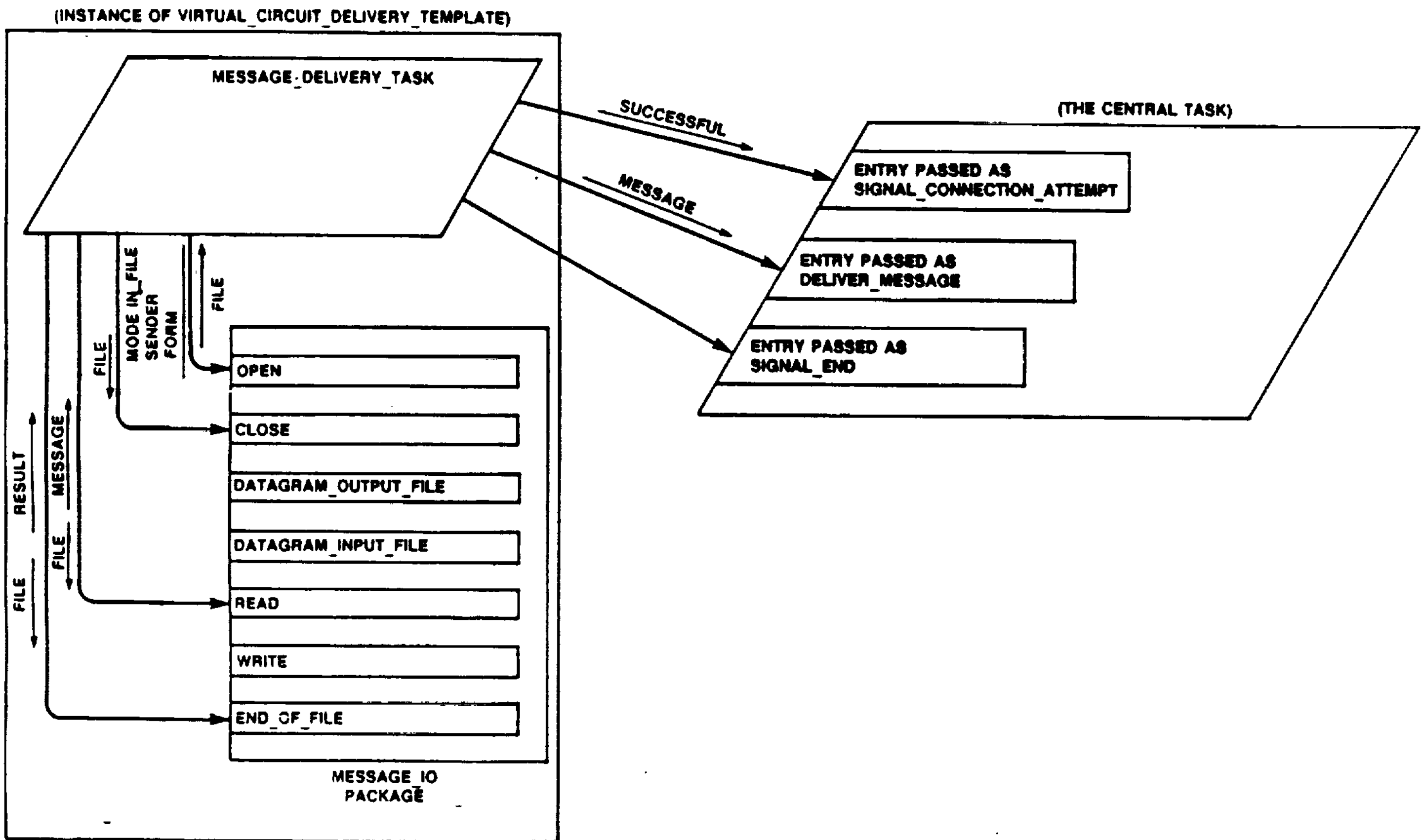
```
generic

   type Message_Type is private;

   Sender : in String;

   Form   : in String;

   with
      procedure Signal_Connection_Attempt
         (Successful : in Boolean);

   with procedure Deliver_Message (Message: in Message_Type);

   with procedure Signal_End;

package Virtual_Circuit_Delivery_Template is

end Virtual_Circuit_Delivery_Template;
```

```
with Network_IO;

package body Virtual_Circuit_Delivery_Template is

   package Message_IO is new Network_IO (Message_Type);

   task Message_Delivery_Task;

      task body Message_Delivery_Task is

         Input_File         : Message_IO.File_Type;
         Message            : Message_Type;
         Connection_Failure : exception;

      begin

         begin
            Message_IO.Open
               (Input_File, Message_IO.In_File, Sender, Form);
         exception
            when others =>
               Signal_Connection_Attempt (Successful => False);
               raise Connection_Failure;   -- Abandon this task.
         end;

         Signal_Connection_Attempt (Successful => True);

         while not Message_IO.End_Of_File (Input_File) loop
            Message_IO.Read (Input_File, Message);
            Deliver_Message (Message);
         end loop;

         Message_IO.Close (Input_File);
         Signal_End;

      end Message_Delivery_Task;

   end Virtual_Circuit_Delivery_Template;
```

Figure 5-7 shows the resulting program structure.

An almost identical generic package can be used to generate a call on a specified entry every time a datagram arrives on a specified stream. In this case, generic parameters specify the type of the datagram, the name of the 5-12 incoming datagram stream, and the entry to be called every time a datagram arrives on the stream:

```
generic

   type Message_Type is private;
   Stream_Name : in String;
   with procedure Deliver_Message (Message: in Message_Type);

package Datagram_Delivery_Template is

end Datagram_Delivery_Template;
```



Entries of the central task are passed as generic parameters to the instantiation of VIRTUAL_CIRCUIT_DELIVERY_TEMPLATE. The entry corresponding to DELIVER_MESSAGE is called every time a message is received, to pass the message to the central task.

Figure 5-7. Program structure using Virtual_Circuit_Delivery_Template

```
with Network_IO;

package body Datagram_Delivery_Template is

   package Message_IO is new Network_IO (Message_Type);

   task Message_Delivery_Task;

   task body Message_Delivery_Task is
      Message : Message_Type;
   begin
      loop
         Message_IO.Read
            (Message_IO.Datagram_Input_File (Stream_Name),
             Message);
         Deliver_Message (Message);
      end loop;
   end Message_Delivery_Task;

end Datagram_Delivery_Template;
```

Figure 5-8 shows the resulting program structure.

The remaining examples are based on Virtual_Circuit_Delivery_Template and Datagram_Delivery_Template, and depict their use. Because these templates are so useful for interleaved processing, we expect that they will be provided as part of the NOS library.

5.5 Merging Streams of Incoming Messages

Processing entity ONE_KW_CELL_MANAGER manages a set of 1-kilowatt power cells that are dynamically switched on-line and off-line in groups. Processing entity FIVE_KW_CELL_MANAGER does the same for a set of 5-kilowatt power cells. Both these processing entities have virtual-circuit connections established, by prior agreement, with a third processing entity, POWER_SUPPLY_MONITOR. Every time ONE_KW_CELL_MANAGER or FIVE_KW_CELL_MANAGER switches a group of cells on or off, a message giving the number of cells switched is sent to POWER_SUPPLY_MONITOR. A positive number indicates that cells were switched on-line and a negative number indicates that cells were switched off-line. POWER_SUPPLY_MONITOR may also receive datagrams in stream REQUEST_STREAM requesting the total wattage of power cells currently on-line, in which case it sends a reply to the return address given in the datagram.

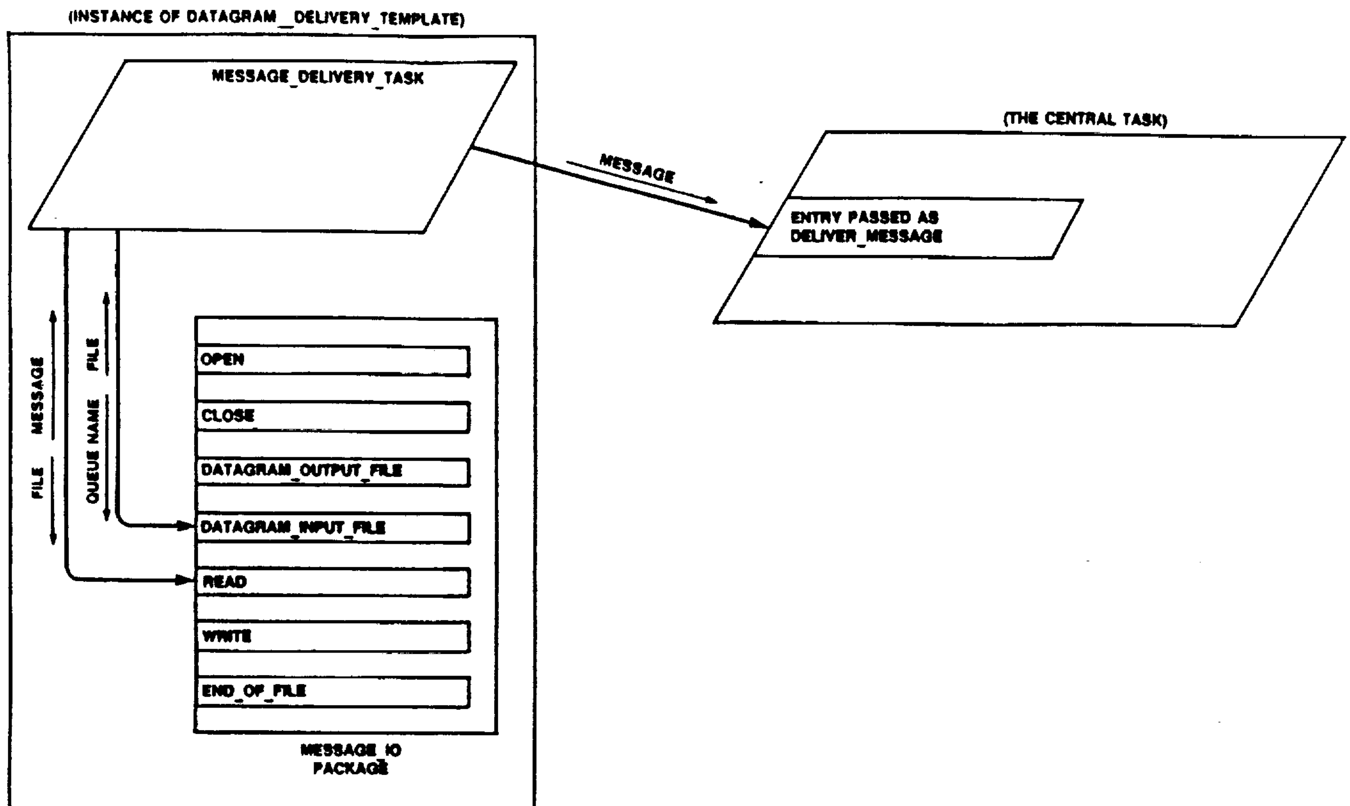


Figure 5-8. Program structure using Datagram_Delivery_Template

As shown in Figure 5-9, POWER_SUPPLY_MONITOR must, in effect, merge three streams of inputs -- a stream of notices about groups of 1-kilowatt cells, a stream of notices about groups of 5-kilowatt cells, and a stream of requests for totals -- into a single stream, to treat each incoming message as it arrives. This can be done by a task with three entries, one for each input stream. One of these entries will be called whenever a message arrives in the corresponding input stream. The task will execute a selective wait for calls on all three entries, thus processing incoming messages as they arrive.

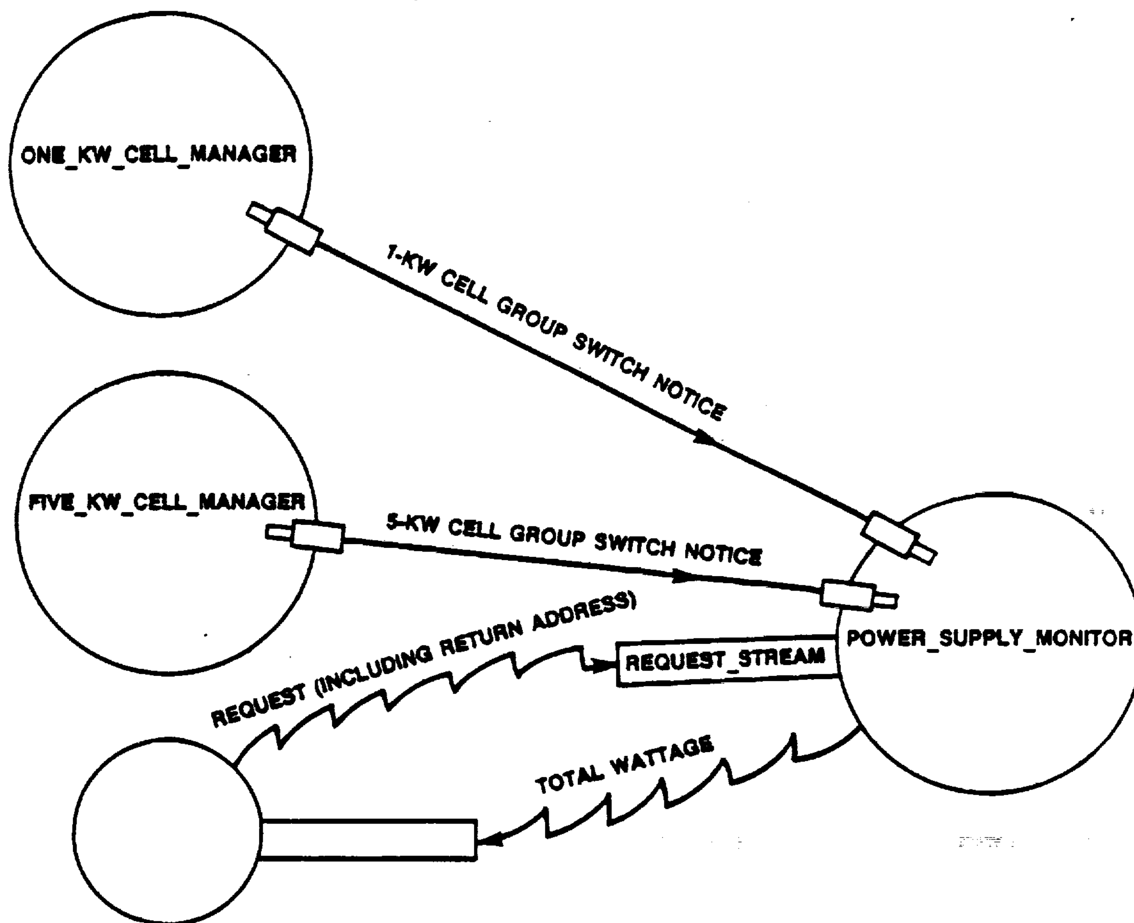


Figure 5-9. Flow of MESSAGES to and from POWER_SUPPLY_MONITOR

The requests for total wattage are messages of the type provided by the following package:

```
package Request_Package is

   type Request_Type is
      record
         Requestor_Part : String (1 .. 128);
         Stream_Part    : String (1 .. 64);
      end record;

   end Request_Package;
```

The work of POWER_SUPPLY_MONITOR is done by the task Monitor_Task. Instances of Virtual_Circuit_Delivery_Template and Datagram_Delivery_Template are used to link the incoming virtual circuits and REQUEST_STREAM to entries of Monitor_Task. The resulting program unit structure is shown in Figure 5-10.

```
   procedure Null_Procedure is
   begin
      null;
   end Null_Procedure;

   package One_KW_Message_Package is
      new Virtual_Circuit_Delivery_Template
         (Message_Type      => Integer,
          Sender            => "ONE_KW_CELL_MANAGER",
          Form              => [some string],
          Signal_Connection_Attempt =>
            Monitor_Task.Handle_Connection,
         Deliver_Message    => Monitor_Task.Handle_1_KW_Group,
         Signal_End         => Null_Procedure);

   package Five_KW_Message_Package is
      new Virtual_Circuit_Delivery_Template
         (Message_Type      => Integer,
          Sender            => "FIVE_KW_CELL_MANAGER",
          Form              => [some string],
          Signal_Connection_Attempt =>
            Monitor_Task.Handle_Connection,
          Deliver_Message => Monitor_Task.Handle_5_KW_Group,
          Signal_End      => Null_Procedure);

   package Request_Package is
      new Datagram_Delivery_Template
         (Message_Type    => Request_Package.Request_Type,
          Deliver_Message => Monitor_Task.Handle_Request);

   package Integer_IO is new Network_IO (Integer);

   task body Monitor_Task is separate;

begin

   null;  -- All work done by tasks.

end Power_Supply_Monitor;


separate (Power_Supply_Monitor)

task Monitor_Task is

   Connection_Failure : exception;
   Total_Wattage      : Integer := 0;
```


PRECEDING PAGE BLANK NOT FILMED

```
   begin

      for I in 1 .. 2 loop
         accept Handle_Connection (Successful : in Boolean) do
            if not Successful then
               raise Connection_Failure;
            end if;
         end Handle_Connection;
      end loop;

      loop

         select

            accept Handle_1_KW_Group (Cell_Count : in Integer) do
               Total_Wattage := Total_Wattage + Cell_Count;
            end Handle_1_KW_Group;

         or

            accept Handle_5_KW_Group (Cell_Count : in Integer) do
               Total_Wattage := Total_Wattage + 5 * Cell_Count;
            end Handle_5_KW_Group;

         or

            accept Handle_Request
               (Request : in Request_Package.Request_Type) do

               Integer_IO.Write
                  (Integer_IO.Datagram_Output_File
                     (Request.Requestor_Part,
                      Request.Stream_Part),
                   Total_Wattage);

            end Handle_Request;

         end select;

      end loop;

   end Monitor_Task;
```

Since the virtual-circuit connections are intended to remain in existence permanently, there is no need to provide an entry to be called upon end of file. The dummy procedure Null_Procedure is used as a generic actual parameter in place of such an entry. The two instances of Virtual_Circuit_Delivery_Template specify the same entry to be called upon

connection establishment. Thus the entry Handle_Connection is called twice when Power_Supply_Monitor starts up. The <u>for</u>-loop at the beginning of the Monitor_Task body handles these calls.

5.6 Processing Datagrams of Different Priority

A processing entity is to process bulletins of three different priorities. Bulletins are values of some type Bulletin_Type and are processed by calling the library procedure Process_Bulletin. A bulletin should not be selected for processing while a higher-priority bulletin is waiting.

Our solution is to establish three different datagram streams, HIGH_PRIORITY_STREAM, MEDIUM_PRIORITY_STREAM, and LOW_PRIORITY_STREAM, corresponding to the three different priority levels of bulletins. This arrangement is illustrated in Figure 5-11. We declare a task with one entry corresponding to each priority level.

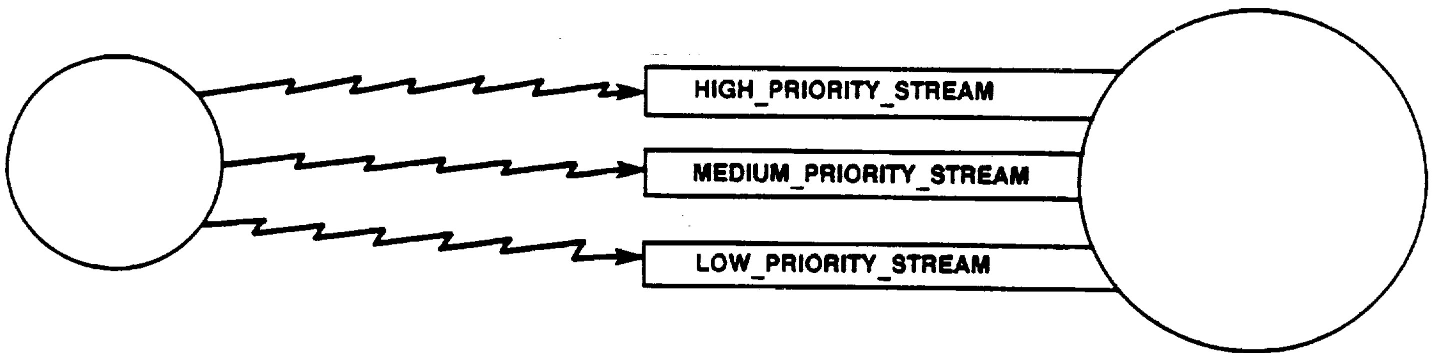


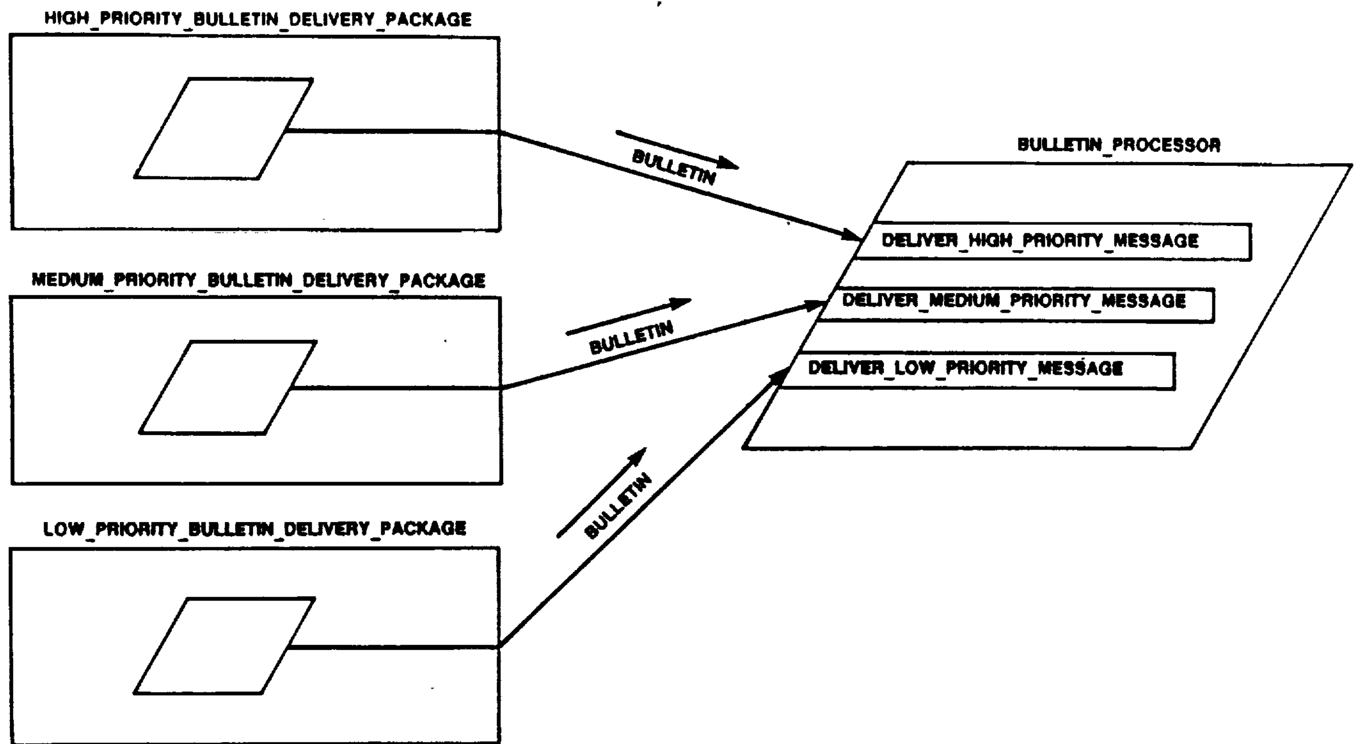
Figure 5-11. A Processing Entity with Datagram Streams Corresponding to Different Priority Levels

```
task Bulletin_Processor is

   entry Deliver_High_Priority_Bulletin
      (Bulletin : in Bulletin_Type);

   entry Deliver_Medium_Priority_Bulletin
      (Bulletin : in Bulletin_Type);

   entry Deliver_Low_Priority_Bulletin
      (Bulletin : in Bulletin_Type);

end Bulletin_Processor;
```

The following generic instantiations ensure that the appropriate entry is called each time a datagram arrives at one of the streams:

```
package High_Priority_Bulletin_Delivery_Package is
   new Datagram_Delivery_Template
      (Message_Type => Bulletin_Type,
       Stream_Name  => "HIGH_PRIORITY_STREAM",
       Deliver_Message =>
         Bulletin_Processor.Deliver_High_Priority_Message);

package Medium_Priority_Bulletin_Delivery_Package is
   new Datagram_Delivery_Template
      (Message_Type => Bulletin_Type,
       Stream_Name  => "MEDIUM_PRIORITY_STREAM",
       Deliver_Message =>
         Bulletin_Processor.Deliver_Medium_Priority_Message);

package Low_Priority_Bulletin_Delivery_Package is
   new Datagram_Delivery_Template
      (Message_Type => Bulletin_Type,
       Stream_Name  => "LOW_PRIORITY_STREAM",
       Deliver_Message =>
         Bulletin_Processor.Deliver_Low_Priority_Message);
```

The resulting program structure is shown in Figure 5-12.



High_Priority_Bulletin_Delivery_Package, Medium_Priority_Bulletin_-Delivery_Package, and Low_Priority_Bulletin_Delivery_Package are instances of the generic package Datagram_Delivery_Template. Each contains a task that calls the corresponding entry of Bulletin_Processor when a datagram arrives on the corresponding stream.

Figure 5-12. Task structure for processing bulletins of different priorities

The Bulletin_Processor task body is a straightforward application of the selective wait statement with an else part:

```
task body Bulletin_Processor is

begin

   loop

      select

         -- First check for a high-priority bulletin.

         accept Deliver_High_Priority_Bulletin
            (Bulletin : in Bulletin_Type) do

            Process_Bulletin (Bulletin);

         end Deliver_High_Priority_Bulletin;

      else

         -- No high-priority bulletin is waiting to be processed,
         --    so consider medium- and low-priority bulletins.

         select

            -- First check for a medium-priority bulletin.

            accept Deliver_Medium_Priority_Bulletin
               (Bulletin : in Bulletin_Type) do

               Process_Bulletin (Bulletin);

            end Deliver_Medium_Priority_Bulletin;

         else

            -- No medium-priority bulletin is waiting to be
            --    processed, so wait for the arrival of the first
            --    bulletin of any priority.

            select

               accept Deliver_High_Priority_Bulletin
                  (Bulletin : in Bulletin_Type) do

                  Process_Bulletin (Bulletin);

               end Deliver_High_Priority_Bulletin;
```

```
            or

               accept Deliver_Medium_Priority_Bulletin
                  (Bulletin : in Bulletin_Type) do

                  Process_Bulletin (Bulletin);

               end Deliver_Medium_Priority_Bulletin;

            or

               accept Deliver_Low_Priority_Bulletin
                  (Bulletin : in Bulletin_Type) do

                  Process_Bulletin (Bulletin);

               end Deliver_Medium_Priority_Bulletin;

            end select;

         end select;

      end select;

   end loop;

end Bulletin_Processor;
```

There are many ways to achieve priority-driven processing of incoming messages. The choice of datagrams for this example was arbitrary. We could just as easily have established three virtual circuits corresponding to the three bulletin priority levels and applied the same approach. The next example illustrates a general scheme to use message contents to determine the order in which messages are processed. Since these message contents could include priority levels, the next example provides an alternative scheme for achieving priority-driven processing of messages.

5.7 Using Message Contents to Control Order of Processing

A warning-system application has an incoming stream WARNING_STREAM for datagrams of the following type:

```
type Warning_Type is
   record
      Category_Part : Category_Type;
      Contents_Part : Contents_Type;
   end record;
```

Category_Type is defined as follows:

```
type Category_Type is
   (From_Ground, From_Other_Station, From_This_Station);
```

The definition of Contents_Type is irrelevant to this example.

The application requires a way to retrieve the next incoming datagram with a particular Category_Part value, or to determine that no such datagram yet exists. This capability will be supplied by the following task:

```
task Warning_Retriever is

   entry Retrieve_Warning (Category_Type)
      (Contents : out Contents_Type);

   entry Deliver_Warning (Warning : in Warning_Type);

end Warning_Retriever;
```

Retrieve_Warning is an __entry family__ with one member for each Category_Type value. A call on the entry family member Retrieve_Warning (From_Ground), for example, waits if necessary for a datagram with a Category_Part value of From_Ground to arrive, then delivers the datagram's Contents_Part. To avoid waiting if a datagram of the appropriate category has not arrived, the application can use a __conditional entry call__:

```
select
   Retrieve_Warning (From_Ground) (Contents);
   Warning_Found := True;
else
   Warning_Found := False;
end select;
```

The entry Deliver_Warning is not used by the application, but is used as part of the implementation of the Warning_Retriever task, as shown in Figure 5-13. The instantiation

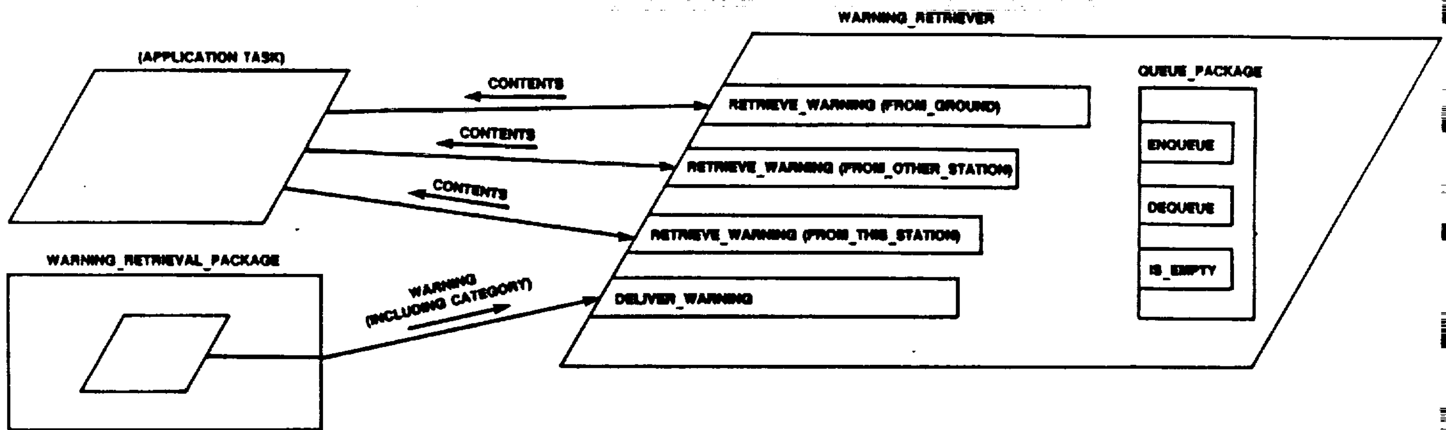
```
package Warning_Retrieval_Package is
   new Datagram_Delivery_Template
      (Message_Type    => Warning_Type,
       Stream_Name     => "WARNING_STREAM",
       Deliver_Message => Warning_Retriever.Deliver_Warning);
```

causes the Deliver_Warning entry to be called with a Warning_Type value every time a datagram arrives on WARNING_STREAM.

To implement the Warning_Retriever task, we assume the existence of a generic package for first-in-first-out queues:

```
generic

   type Element_Type is private;

package Generic_Queue_Package is

   type Queue_Type is limited private;
      -- Default initial value is an empty queue.

   procedure Enqueue
      (Queue : in out Queue_Type; Item : in Element_Type);

   procedure Dequeue
      (Queue : in out Queue_Type; Item : out Element_Type);

   function Is_Empty (Queue : Queue_Type) return Boolean;

   Empty_Queue_Error : exception;

private

   type Queue_Type is [some type definition];

end Generic_Queue_Package;
```

This is a template for a typical Ada package providing an abstract data type and a set of operations for manipulating that type.



Warning_Retrieval_Package is an instance of Datagram_Delivery_Template. It contains a task that calls Warning_Retriever.Deliver_Warning every time a datagram arrives on WARNING_STREAM. Internal to the Warning_Retriever task, Queue_Package is an instance of Generic_Queue_Package, used to maintain a separate queue for each category of warning.

Figure 5-13. Use of the Warning_Retriever Task to Control the Order in Which Messages are Processed

The body of Warning_Retriever is as follows:

```
task body Warning_Retriever is

   package Queue_Package is
      new Generic_Queue_Package (Contents_Type);

   use Queue_Package;

   Queue_Table : array (Category_Type) of Queue_Type;

begin

   loop

      select

         accept Deliver_Warning (Warning : in Warning_Type) do

            Enqueue
               (Queue_Table (Warning.Category_Part),
                Warning.Contents_Part);

         end Deliver_Warning;

      or

         when not Is_Empty (Queue_Table (From_Ground)) =>

            accept Retrieve_Warning (From_Ground)
               (Contents : out Contents_Type) do

               Dequeue (Queue_Table (From_Ground), Contents);

            end Retrieve_Warning;

      or

         when not Is_Empty (Queue_Table (From_Other_Station)) =>

            accept Retrieve_Warning (From_Other_Station)
               (Contents : out Contents_Type) do

               Dequeue
                  (Queue_Table (From_Other_Station), Contents);

            end Retrieve_Warning;
```

```
or

   when not Is_Empty (Queue_Table (From_This_Station)) =>

      accept Retrieve_Warning (From_This_Station)
         (Contents : out Contents_Type) do

         Dequeue
            (Queue_Table (From_This_Station), Contents);

      end Retrieve_Warning;

   end select;

end loop;

end Warning_Retriever;
```

Figure 5-13 illustrates the relationship of the various program units.

5.8 Using Datagrams to Control Periodic Sampling

A processing entity SENSOR_MANAGER is responsible for providing periodic sensor data to other processing entities upon request. SENSOR_MANAGER can service at most one other processing entity at a time. Service can be requested by sending a datagram to CONNECTION_STREAM. The processing entity currently being serviced can send subsequent datagrams to INTERVAL_STREAM requesting a change in sampling interval or to DISCONNECTION_STREAM releasing SENSOR_MANAGER to serve other processing entities. For each datagram received on CONNECTION_STREAM, SENSOR_MANAGER sends an acknowledgment datagram of type Boolean, equal to True if SENSOR_MANAGER is available and False if SENSOR_MANAGER is busy. If SENSOR_MANAGER is available, a virtual-circuit connection is established to send sensor readings to the requestor. See Figure 5-14.

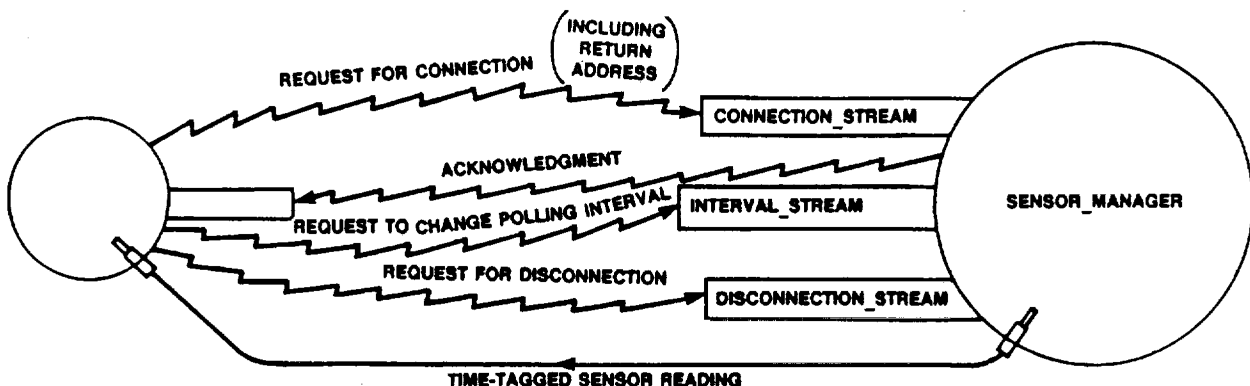


Figure 5-14. Network communication with SENSOR_MANAGER

The following package declares the type of datagrams sent to CONNECTION_STREAM:

```
with Calendar;

package Service_Request_Package is

   type Service_Request_Type is
      record
         Requestor_Name             : String (1 .. 64);
         Acknowledgment_Stream_Name : String (1 .. 64);
         Initial_Interval           : Calendar.Duration;
      end record;

end Service_Request_Package;
```

Datagrams sent to INTERVAL_STREAM are values of type Calendar.Duration and datagrams sent to DISCONNECTION_STREAM belong to a record type with no components. (The simple existence of a message conveys the required information.) Sensor readings are time-tagged values of type Float, as defined in the following package:

```
with Calendar;

package Time_Tagged_Float_Package is

   type Time_Tagged_Float_Type is
      record
         Data_Part : Float;
         Tag_Part : Calendar.Time;
      end record;

end Time_Tagged_Float_Package;
```

These values are read by a library procedure Sample_Sensor and sent over the virtual circuit.

We introduce a new library generic package Null_Datagram_Delivery_Template. This package is similar in form and purpose to the generic package Datagram_Delivery_Template introduced earlier in Section 5-4. However, Null_Datagram_Delivery_Template is specially tailored to the receipt of empty datagrams like those to be sent on DISCONNECTION_STREAM. The entry called upon receipt of a message has no parameters. Use of Null_Datagram_Delivery_Template instead of Datagram_Delivery_Template avoids the work of passing a dummy entry parameter that contains no information. Here is the text of Null_Datagram_-Delivery_Template:

```
generic

   Stream_Name : in String;
   with procedure Signal_Message;

package Null_Datagram_Delivery_Template is

end Null_Datagram_Delivery_Template;
```

```
package body Null_Datagram_Delivery_Template is

   type Null_Type is record null; end record;

   task Message_Delivery_Task;

   package Null_Message_IO is new Network_IO (Null_Type);

   task body Message_Delivery_Task is
      Message : Null_Type;
   begin
      loop
         Null_Message_IO.Read
            (Null_Message_IO.Datagram_Input_File
               (Stream_Name),
             Message);
         Signal_Message;
      end loop;
   end Message_Delivery_Task;

end Null_Datagram_Delivery_Template;
```

Here is the SENSOR_MANAGER program:

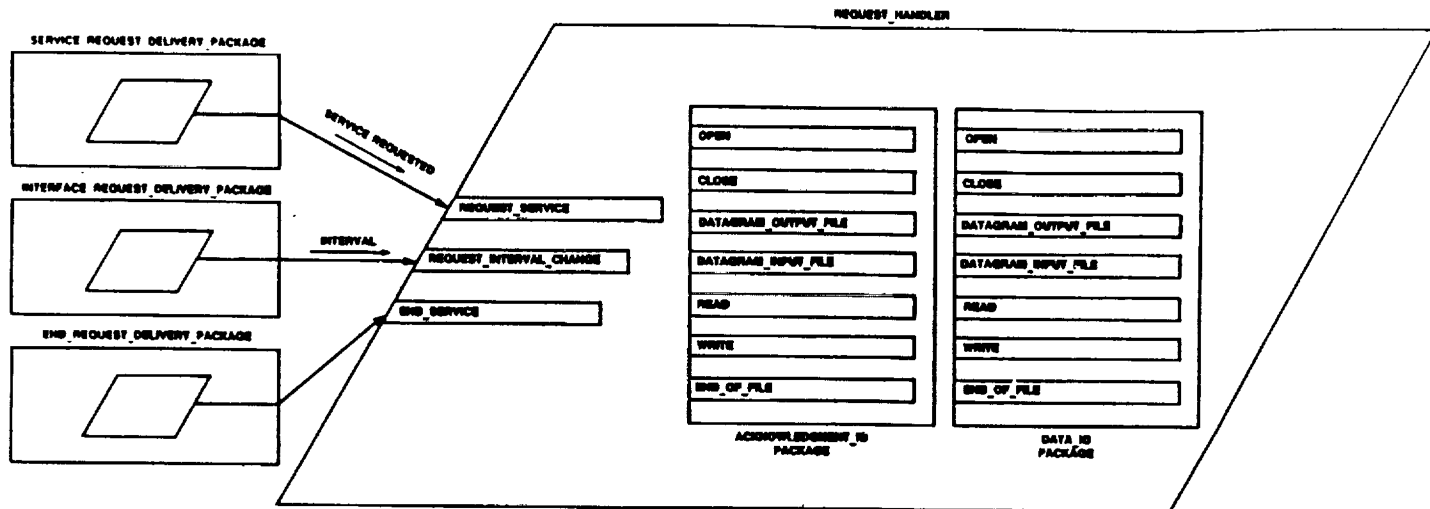
```
with Service_Request_Package, Calendar, Network_IO;

procedure Sensor_Manager is

   subtype Service_Request_Type is
      Service_Request_Package.Service_Request_Type;

   task Request_Handler is
      entry Request_Service (Request : in Service_Request_Type);
      entry Request_Interval_Change
         (New_Interval : in Calendar.Duration);
      entry End_Service;
   end Request_Handler;

   task body Request_Handler is separate;
```

```
   package Service_Request_Delivery_Package is
      new Datagram_Delivery_Template
         (Message_Type    => Service_Request_Type,
          Stream_Name     => "CONNECTION_STREAM",
          Deliver_Message => Request_Handler.Request_Service);

   package Interval_Request_Delivery_Package is
      new Datagram_Delivery_Template
         (Message_Type    => Calendar.Duration,
          Stream_Name     => "INTERVAL_STREAM",
          Deliver_Message =>
            Request_Handler.Request_Interval_Change);

   package End_Request_Delivery_Package is
      new Null_Datagram_Delivery_Template
         (Stream_Name    => "DISCONNECTION_STREAM",
          Signal_Message => Request_Handler.End_Service);

begin

   null;  -- All work done by tasks.

end Sensor_Manager;

with Time_Tagged_Float_Package, Sample_Sensor;
use Calendar;

separate (Sensor_Manager)

task body Request_Handler is

   subtype Data_Type is
      Time_Tagged_Float_Package.Time_Tagged_Float_Type;

   package Acknowledgment_IO is new Network_IO (Boolean);
   package Data_IO is new Network_IO (Data_Type);
   use Acknowledgment_IO, Data_IO;

   Output_File          : Data_IO.File_Type;
   Current_Interval     : Duration;
   Previous_Sample_Time : Time;
   Next_Sample_Time     : Time;
   Data                 : Data_Type;

   Connection_Form : constant String := [some string literal];
```

```
begin

   loop

      accept Request_Service
         (Request : in Service_Request_Type) do

         Open
            (Output_File,
             Out_File,
             Request.Requestor_Name,
             Connection_Form);

         Write
            (Datagram_Output_File
               (Request.Requestor_Name,
                Request.Acknowledgment_Stream_Name),
             True);

         Current_Interval := Request.Initial_Interval;

      end Request_Service;

      Previous_Sample_Time := Clock - Current_Interval;

      loop

         Next_Sample_Time :=
            Previous_Sample_Time + Current_Interval;

         select

            delay Next_Sample_Time - Clock;
            Sample_Sensor (Data);
            Write (Output_File, Data);

         or

            accept Request_Service
               (Request : in Service_Request_Type) do

               Write
                  (Datagram_Output_File
                     (Request.Requestor_Name,
                      Request.Acknowledgment_Stream_Name),
                   False);
                  -- Busy with another processing entity.

            end Request_Service;
```

```
          or

             accept Request_Interval_Change
                (New_Interval : in Duration) do

                 Current_Interval := New_Interval;

             end Request_Interval_Change;

          or

             accept End_Service;
             exit;

          end select;

       end loop;

       Close (Output_File);

    end loop;

 end Request_Handler;
```

The outer loop in the task body is repeated once for every session (consisting of a request for a virtual-circuit connection, a connection followed by transmission of sensor data, zero or more interval changes, and a request for disconnection). The inner loop is repeated once for each datagram that arrives during a session. Figure 5-15 illustrates the program-unit structure.



Acknowledgment_IO (used for acknowledgment datagrams) and Data_IO (used to send time-tagged sensor readings over a virtual-circuit connection) are both instances of Network_IO. Service_Request_Delivery_Package is an instance of Datagram_Delivery_Template that calls Request_Handler.Request_Service whenever a datagram arrives on CONNECTION_STREAM. Interval_Request_Delivery_Package is an instance of Datagram_Delivery_Template that calls Request_Handler.Request_Interval_Change every time an datagram arrives on INTERVAL_STREAM. End_Request_Delivery_Package is an instance of Null_Datagram_Delivery_Template that calls Request_Handler.End_Service every time a datagram arrives on DISCONNECTION_STREAM. Each of these three instances contains an instantiation of Network_IO, not shown here.

Figure 5-15. Program-unit structure for SENSOR_MANAGER

ORIGINAL PAGE IS
OF POOR QUALITY