NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

Design and Implementation of a Communications Subsystem for the Homogeneous Multiprocessor

Walter Prager

A Thesis

in

The Department

of

Computer Science

Presented in Partial Fulfillment of the Requirements
for the Degree of Master of Computer Science at
Concordia University
Montreal, Quebec, Canada

May 1989

ISBN 0-315-51358-6

ABSTRACT

Design and Implementation
of a Communications Subsystem
for the Homogeneous Multiprocessor.

Walter Prager

This thesis describes the design and implementation of the communication layer of an operating system kernel for the Homogeneous Multiprocessor (HM). The HM has a linear-array topology, with interprocess communications achieved through a high-speed, parallel bus as well as by the sharing of memory between nearest neighbours. The communications layer design provides a general framework which allows easy implementation of needed protocols suitable for the many intended uses of the multiprocessor, and simple mechanisms which provide low-overhead access to the various protocol layers. An implementation of the framework and the link-level protocol has been completed.

ACKNOWLEDGEMENTS

The author wishes to express the deepest gratitude to his supervisor, Dr. J. W. Atwood, without whose careful supervision, invaluable help, and inexhaustible patience this work would not have been possible.

This project was supported in part by a Natural Sciences and Engineering Research Council of Canada (NSERC) Operating Grant, by an NSERC Postgraduate Scholarship, and by the Quebec Ministere de l'Education, de la Science, et de la Technologie, through its Action Structurante program.

Dedicated to my wife

Terry Ann

Table of Contents

List of Figures.

List of Appendices.

CHAPTER I.

Introduction.

Due to the technological advances of the last half of this century modern-day computers can deliver more processing power than ever thought possible, yet at a fraction of the cost of their predecessors. Super computers, like the Cray 1, can perform several million floating-point operations per second (MFLOPS). Yet as the processors increased in size and complexity, so did the user community. Computers are now being used by more people for more varied applications than ever before. Demand on performance has never been greater.

While a large mainframe can accommodate most applications satisfactorily, its cost can be rather prohibitive. Also, in many applications the full power of a mainframe (such as floating-point arithmetic or complicated I/O processing) is not needed, while its high speed is still required. Using a large computer in such cases results in wasting much of its processing power.

The advent of the microprocessor and personal computers changed the situation tremendously. It now became possible to provide each user with a dedicated (albeit) small machine at a relatively small cost, when compared with the price of a large mainframe. As microprocessors became more powerful and less expensive (a trend which is still continuing), it became apparent that the combined power of many smaller

machines could match that of a larger mainframe at a lower overall cost.

A better cost-efficiency is not the only reason for connecting processors together. Some applications, such as the control of the life support-system on a space shuttle, require a fault-tolerance which just cannot be achieved on a uniprocessor, however fast. Other applications, such as weather forecasting or flight simulation require the processing to be done in parallel--a feature that can only be simulated with a single processing element. Network-type connections allow sharing of expensive and relatively rarely-used resources, such as printers. Diskless workstation configurations sharing a file server provide users with a large central file system as well as a dedicated processor.

The concept of connecting many small processors to achieve greater performance is used even in the construction of mainframe computers. Pipelined architectures and vector processors use an interconnection of simple processing elements to derive incredible overall power.

While physical connectivity lets data be transferred between processors, software support is required to allow this feature to be used as a tool for process cooperation. In fact, the success or failure of a multiprocessor to achieve desired processing power depends almost as much on the design of the communication subsystem as it does on its architecture. The communication layer can be viewed as a

bridge between the hardware interconnection and the requirements of the intended applications. Some of the functions of this layer could include process addressing, error detection and correction, preserving message boundaries and so on.

Multiple-processor architectures can be classified according to several schemes. One classification is based on the type of data and instruction streams used--single or multiple. A typical uniprocessor would be classified as a Single Instruction stream/Single Data stream (SISD) machine. A vector processor would fall under the Single Instruction stream/Multiple Data stream (SIMD) designation. A true "multiprocessor"--a machine capable of carrying out several related or unrelated tasks simultaneously--would then be termed a Multiple Instruction stream/Multiple Data stream (MIMD) architecture.

Another method of classifying multiple processor architectures is by the degree of coupling between the component elements. The degree of coupling is related to both the physical connections between the processors and the logical relationship between the components. For example, an array-processor is a very-tightly coupled architecture, since the processors cooperate at the instruction level. A Wide Area Network (WAN) or a resource-sharing environment would be considered loosely-coupled. If any cooperation is present it is probably at the user level. In addition, the communication medium in WANs is slow in comparison with other forms

of communication, and the distances between the processors are generally large. A Local Area Network (LAN) where some task-distribution is present (e.g., Remote Procedure Calls--RPC) would fall somewhere in between the first two examples.

As far as the methods of communication are concerned, they are largely determined by the intended application. Close cooperation between the processors would require that the supporting communication medium to be reasonably fast. The closer the cooperation, the faster the communication speed required. Thus satellite transmissions and other WANs generally indicate a loosely-coupled system, while shared memory and common-bus interconnections would suggest a tightly-coupled architecture.

The Homogeneous Multiprocessor, [Dimo83, Dimo87, Li87a, Li87b], is an architecture currently being developed by Dr. Nikitas Dimopoulos. It is a "true" multiprocessor in the sense that it is an MIMD architecture, as defined above. The multiprocessor is structured as a linear array of identical processing elements (hence its name) with communications supported by a high-speed LAN as well as memory sharing between neighboring processors. The architecture is intended to support general-purpose applications, although it should be noted that it is especially well suited to support pipeline algorithms commonly used in pattern-recognition, digital system processing and neural network simulation.

The overall goal of the current project was to design and implement the Communication Subsystem for the operating system of the Homogeneous Multiprocessor. Such a subsystem has to utilize the available resources (shared memory, fast network) to provide an efficient and orderly data-transfer facility between the component processors. This system has to be efficient enough to accommodate the requirements of time-dependent applications (such as real-time processing of speech or image data), yet flexible enough to allow implementation of various higher-level protocols needed to support such applications as distributed databases, as well as the more immediate task of supplying file-transfer and loading facilities to the multiprocessor.

The rest of this report is organized as follows. Chapter II discusses the structure of the Homogeneous Multiprocessor, both its architecture and the operating system kernel implemented for it. Chapter III presents various existing communication protocols in the context of the requirements of the project. Chapter IV presents the design of the Communications Subsystem. Chapter V describes STREAMS--a communication subsystem for the Unix[1] operating system--which became the chosen method for implementation. Chapter VI contains a detailed description of the implementation environment and the implementation itself. Chapter VII presents an overview and conclusions of the Thesis.

---

[1]Unix is a trademark of Bell Laboratories.

CHAPTER II.

The Homogeneous Multiprocessor.

This chapter presents the structure of the Homogeneous Multiprocessor. Readers who are already familiar with this subject may proceed to the following chapter. A brief overview is given in the next paragraph for those who wish only a general introduction of the topic or to refresh already-known information.

The Homogeneous Multiprocessor [Dimo83, Dimo85, Dimo87, Li87a, Li87b] is a tightly-coupled architecture composed of a linear array of processors. The processors are connected by a high-speed parallel LAN, and each has the ability to access the memories of its nearest neighbors. The latter feature is made possible by dynamically fusing two local busses into an "extended" bus, by the closing of an intervening switch. The operating system kernel for the multiprocessor--the HM-Nucleus--provides interprocess communication support to user programs in the form of Shared Regions, Global Memory, Remote Procedure Calls, Pipes, User Channels, and Remote Signalling. The Shared Regions provide a method for a two or three-way sharing of data, as well as the mechanisms necessary for mutually exclusive access to the shared data. Global Memory emulates a globally-visible

memory. Pipes support one-to-one or broadcast mode communication between processes[1].

1. The Homogeneous Multiprocessor Architecture.

As shown in Figure 1, the topology of the HM is a linear array of processors. A complete system is composed of k ($k \geq 3$) processors, k memory modules, k+1 interbus switches and the H-Network--a fast, parallel LAN. Logically the structure is divided into two parts: the Homogeneous Multiprocessor Proper and the H-Network.

The Homogeneous Multiprocessor Proper.

The HM Proper comprises the processing elements, memory modules and interbus switches.

The processors used are standard, off-the-shelf Motorola MC68000 microprocessors, and operate independently from each other. Memory access is controlled by the MC68451 memory management unit (MMU), and the access to the H-Network is controlled by the Network Station, designated as HS in Figure 1. Each processor, P_i, owns its local memory module, M_i, which is accessed via the local bus, b_i. It also has exclusive use of the network station, HS_i, which controls the access to the H-Network.

In addition to the H-Network, each processor can access the memory modules of the neighbors to its immediate left

---

[1]A pipe in the HM-Nucleus denotes a communication method similar in concept to Unix pipes, but extended to cover interprocessor communications.

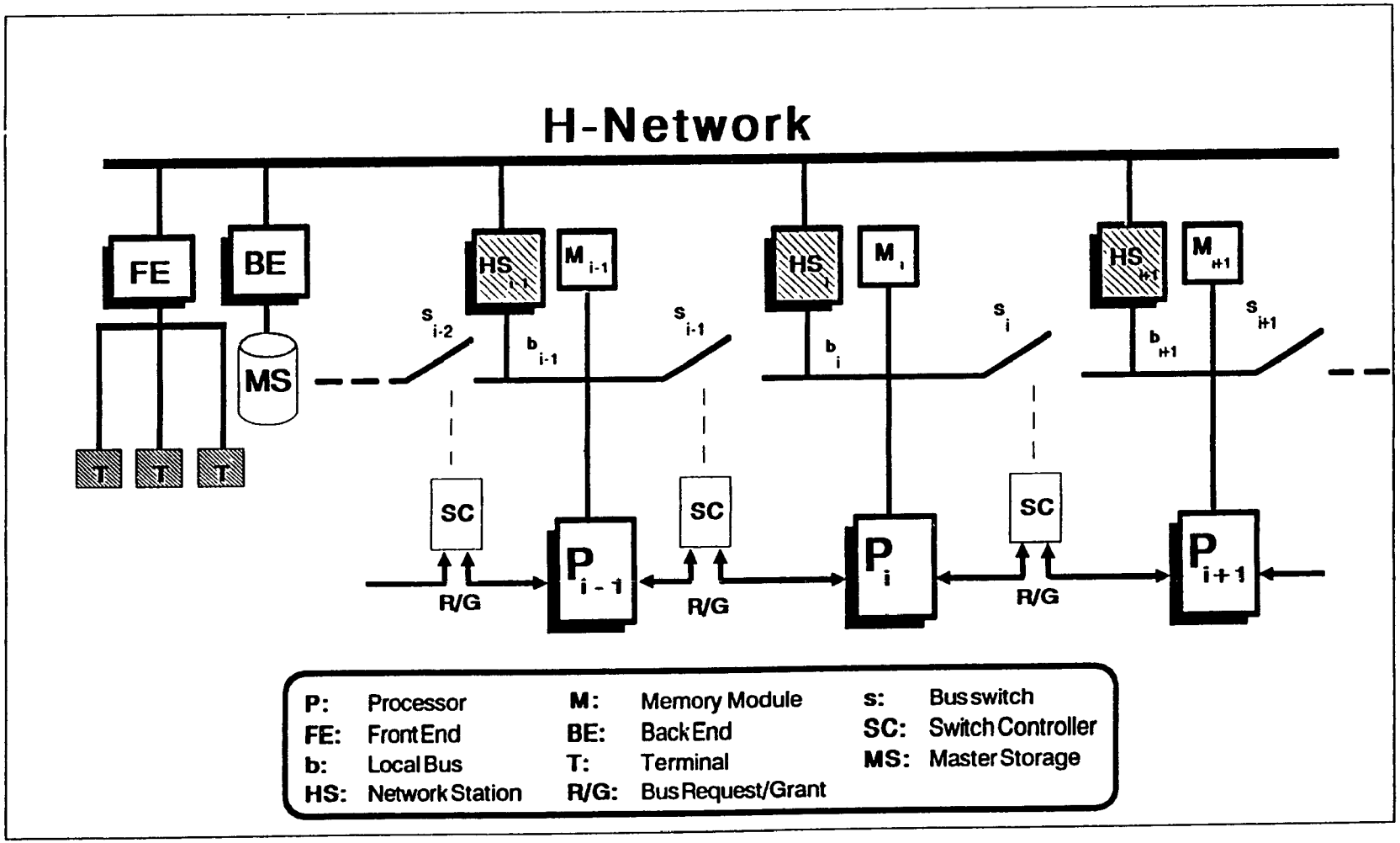

Figure 1. The Homogeneous Multiprocessor Architecture

and right via the "extended bus" feature. An extended bus is created when local busses b_i and b_{i+1} are dynamically fused by the closing of the interbus switch, s_i. The closing of the switch is triggered by a request from one of the processors having access to it (P_i or P_{i+1}), and regulated by a special algorithm which prevents deadlock and ensures liveness. The algorithm prevents two adjacent switches from closing at the same time, while ensuring that any request to close a switch is granted within a finite period of time. The algorithm is presented in Appendix A.

After the physical closing of the switch, the extended bus exists for the duration of the request. After this time (typically one memory cycle) the switch is opened and the extended bus decomposes into its component local busses.

The address space of each processor is divided into four parts: local memory; left and right shared memories; and local unshared memory. An access to the left (or right) shared memory is mapped onto a corresponding address in the memory module of the left (or right) neighbor and carried out via the extended bus. Typically, a remote memory access takes two to three times as long as a local one, depending on the frequency of access. This is due to the time taken by the closing of the switches (both the execution of the algorithm and the physical closing), which increases if there is contention with the processor adjacent to the switch being closed.

The H-Network.

While the extended bus mechanism allows direct communication only between adjacent processors (a multi-hop protocol would allow messages to propagate via the shared memory between non-neighboring processors), the H-Network is well suited for communications between distant processors as well as for broadcasting. It can also support data transfer to and from a back- or front-end processor, which will be used for program development, initial loading and, eventually, for file system support and swapping.

The H-Network is a high-speed parallel LAN capable of 14 megabyte (Mb) data transfer rates. It is similar to the Ethernet in structure, but uses word-parallel data transfer as well as separate access and control lines.

The H-Network uses the Carrier Sense Multiple Access (CSMA) protocol to control network acquisition. At any time only one station--the transmitting station--is deemed to be the network master. All other stations can sense the presence of a carrier signal on the line, indicating that a transmission is in progress, and will refrain from using the network until the operation has completed.

With this scheme it is still possible for two or more stations to find the line "free" and attempt to send their data at almost the same time. The common solution is to use a Cllision Detection (CSMA/CD) method with various back-off intervals to resolve contention. Taking Ethernet as an example, a transmitting station will "listen" to the line

while it is transmitting, making sure that the bits being transmitted correspond to the ones it is sending. If a discrepancy is detected, then a collision with another station's packet must have occurred. Each station that detects a collision stops transmitting and backs off or refrains from using the network for a short period of time before attempting to retransmit. The back-off interval can be randomly picked, fixed, or dynamically adjusted, depending on the volume and nature of traffic on the network.

The CSMA/CD scheme works well when the average packet size is large. In this case the aborted portion of the packet is a small fraction of the overall data. However if packets are smaller in size a larger fraction of the data will have to be retransmitted and network efficiency will decrease.

The H-Network uses a variant of CSMA, called Collision Free (CSMA/CF) [Wang85]. This method works well for both small and large packets. A station manifests its need to transmit by placing a request on the Contention Channel. It is possible to differentiate between the presence of one or several requests on this channel and a station will not place its request on an already requested channel. If after a short interval, called the Contention Interval, the station detects only one request on the channel, it is assured that it will be the only one to be granted access. If more than one request is present, the station withdraws its request and backs off, much as in the above case.

The contention channel in the H-Network is one line, and it is separate from the data transmission path. This implies that while a transmission is taking place the remaining stations can contend for the use of the network when it becomes free. Due to the small contention interval there is a high probability that the next network master will be determined by the time the current transmission is terminated. As the name of the scheme suggests, collisions are thus avoided and network efficiency is increased to its maximum.

2. The Homogeneous Multiprocessor Nucleus.

As in many present distributed systems, the design proposed by Li [Li87a] for the operating system for the Homogeneous Multiprocessor (the HM-Nucleus), revolves around the concept of a **kernel**. Simply put, a kernel provides the essential services of an operating system, extending the bare hardware with runtime support for high-level language constructs. These include such services as process and memory management, interprocess communication primitives, low-level I/O and others. Other features of the operating system, such as swapping, the file system and program development tools can then be built on top of the kernel, using the services it provides. The designers of the Homogeneous Multiprocessor use the term "nucleus" to denote kernel, and reserve the word "kernel" to denote the innermost layer of the nucleus.

The HM-Nucleus is structured in a hierarchical fashion, as proposed by Brown, Denning and Tichy [Brow84]. A layered structure has the advantages of information hiding and data encapsulation, as well as enforcing a modular design. This allows various layers to be modified and new ones to be added without requiring changes in the other layers.

Brown et al. proposed fifteen levels of abstraction in their model operating system. From bottom to top they are: electronic circuits, instruction set, procedures, interrupts, primitive processes, local secondary store, virtual memory, capabilities, communications, file system, devices, stream I/O, user processes, directories and shell. The first eight layers, from electronic circuits to capabilities, are referred to as the single-machine levels. Services provided by these layers never cross over processor boundaries. The remaining layers are called the multiple-machine levels as the functions in these layers may affect remote machines in a multiprocessor environment.

The design of the HM-Nucleus deviates somewhat from this order, mainly because Brown's model gave a user's view of the operating system structure, while the HM-nucleus reflects the structure from the point of view of the implementation. In addition, not every node has local secondary store, and two methods of interprocessor communication are present, which forces communications (levels 4-6) to be defined much closer to the hardware than in Brown's model. Also, since the HM-Nucleus is not a complete operating

system, some levels are not included at all. The following paragraphs briefly describe the structure of the HM-Nucleus. A more thorough description can be found in [Li87a].

The HM-Nucleus consists of eleven layers (Figure 2). They are: Kernel, Physical Memory Manager, Device Management, Capabilities, Universal Datagram Services, Remote Procedure Calls, Communications, Virtual Memory Manager, File Management, Table, and User Interface.

The first five layers correspond, roughly, to the single-machine layers (1-8) of Brown's model. The Kernel provides extensions to the hardware such as process switching, interrupt handling and access to the Memory Management Unit.

The Device Management layer extends the Kernel by providing software drivers for the peripheral devices attached to the processors, and for channels, which use the shared memory as a communications medium. In most nodes the only peripheral will be the H-Network station controller, but some specialized nodes will also contain access to disk drives and serial line interfaces (possibly to outside networks). The Kernel and Device layers map onto levels 1-5 of Brown's model.

The Physical Memory Manager uses the Buddy algorithm to allocate both local and shared memory to user processes and the communication layers. This layer also manages the Memory Manager Unit (MMU) Descriptors, which perform virtual

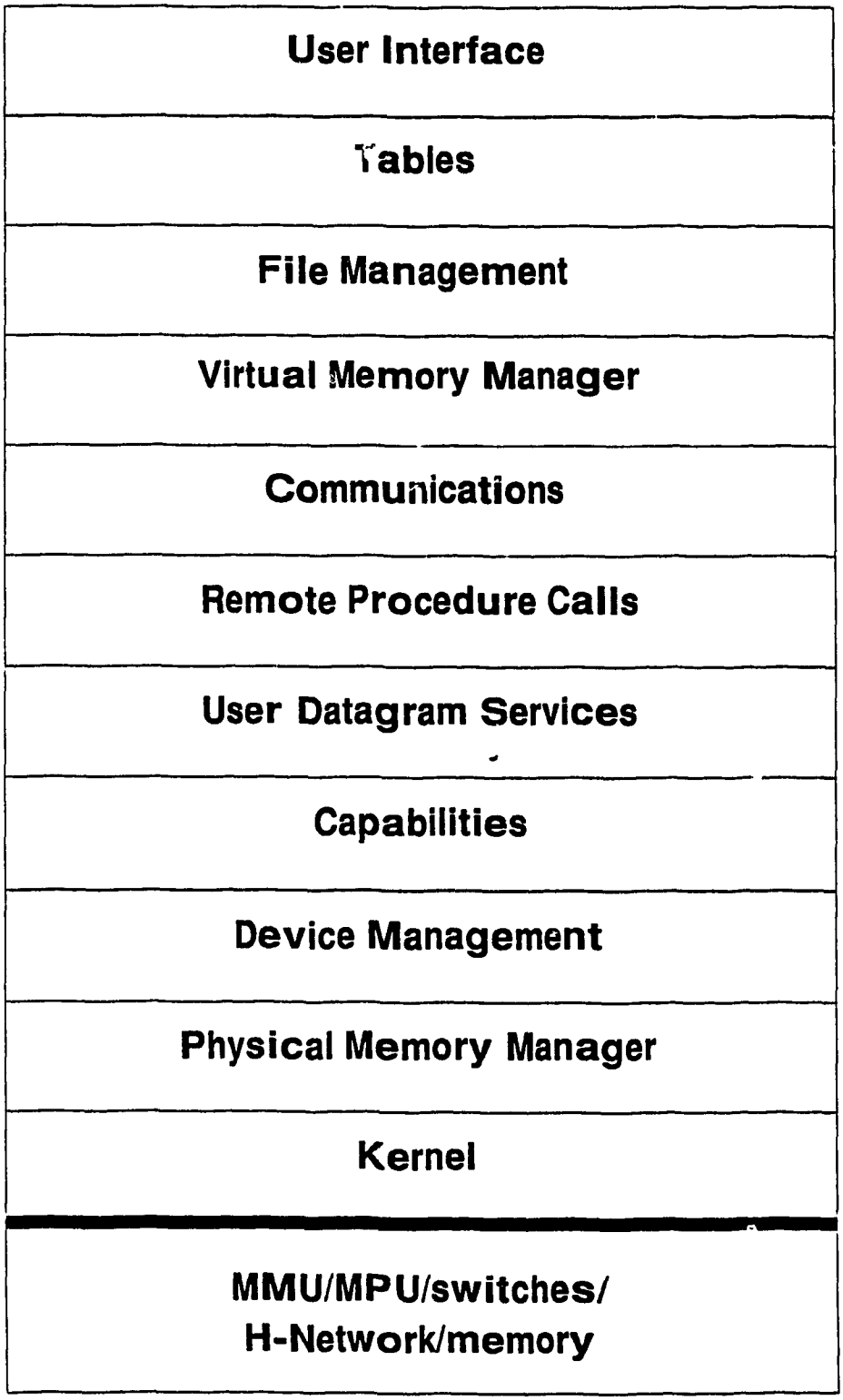

Figure 2. The Structure of the HM-Nucleus

to physical memory mapping and low-level capability checking.

The Capabilities layer provides mechanisms to create and maintain capabilities, which consist of a type, a processor number, a sequence number and a pointer to the information concerning the object described by that capability.

The next three layers--the Universal Datagram Services (UDS), Remote Procedure Calls (RPC), and Communications (COM)--compose the Interprocess Communication Subsystem of the HM-Nucleus.

The UDS layer provides a datagram-oriented service between processes on different machines. In most standard protocols, datagrams are short (their size is constrained by the maximum frame size of the physical media), and no routing is performed. However, the Universal Datagram Service (UDS) as described in [Panz85] supports uniform access to the various communication media accessible by the system, and permits each datagram to be of essentially any size. This is especially relevant in the HM because of the variety of communication paths (H-Network, Shared Memory, Serial lines).

RPC is a higher level protocol built on top of the datagram service. Remote procedure calls are the typical mode of communication in a client/server environment and are suitable for implementing file transfer protocols between the multiprocessor and a front- or back-end processor. The

RPC layer also serves as the interface between the single and multiple machine abstractions.

The COM layer provides communication between user processes, in the form of Pipes, User Channels and a Remote Signalling mechanism. A pipe provides stream-oriented communication on a process-to-process or broadcast basis. The data are transmitted via the H-Net, or via memory buffers that are shared between adjacent processors. Channels allow processes on up to three neighboring machines to exchange a small buffer. The remote signalling mechanism permits a processor P_i to interrupt processors P_{i-2}, P_{i-1}, P_{i+1}, and P_{i+2}.

As the word "channel" is used in [Li87a] to connote both this channel and the Device Management layer channel the terms "user channel" and "channel device" will be used where the distinction is not clear from the context.

The next layer--the Virtual Memory Manager--supports the abstractions of virtual memory space, Shared Regions for user processes, and Global Memory. The communication and synchronization required to implement the latter two abstractions must be provided by the Communications Subsystem. The VMM will eventually include routines to implement swapping by using the UDS over the H-Network.

The File Management layer will implement a hierarchical file system based on that of Unix. Branches of the tree will reside in nodes having a local disk. Other nodes will route their file system requests to these server nodes.

The Table layer, at present, consists only of tables which map symbolic names to capabilities. The tables are not replicated, defining only the capabilities owned by a process on the local node.

Portions of the Nucleus have been implemented in the Concurrent Euclid (CE) language. The kernel for this nucleus was based on the Tunis kernel. Tunis is a Unix-like operating system implemented in CE, and it provides the runtime support for CE primitives.

3. Features to Support Communications.

As stated previously, the H-Network has a structure similar to that of the Ethernet. The basic unit of data transfer is a packet, which can be addressed to an individual network station or sent in broadcast mode. The length of the packet is limited by the controller hardware (the present implementation allows a 128 byte packet, including all header and trailer overhead). This means that the UDS layer will have to incorporate code to handle fragmentation and additional (process) addressing to support an interprocess datagram service.

Communication using the H-Network does not require any original features from the operating system or from the hardware. Due to its similarity to the Ethernet, a driver for the H-Network could be developed relatively easily, using an existing driver as a model. The IEEE 802.3 standard (see chapter III) or the Ethernet protocol can be used

as the Physical level protocol with few, if any, modifications. This has the advantage that a driver for an actual Ethernet controller could be introduced with minimal effort.

Communication using the shared memory, however, is not as straightforward. The hardware which implements the switch closing algorithm is a custom design [Sega88] and no standards exist for communicating via shared-memory connections. The following paragraphs outline some of the features of the HM-Nucleus that were proposed by Li [Li87b], to support shared memory communications. Some slightly different proposals will be mentioned in later chapters.

In support of the shared memory transfers, the Device layer implements "channels"[2]. Briefly, a channel device is a small buffer whose location is fixed at start-up time and well known to one of the neighboring processors. Separate right and left channel devices are allocated to transfer short messages via the shared memory.

Virtual channels are defined to support additional addressing. User processes do not use these virtual channels directly; the additional addressing would be analogous to the process addressing in the UDS layer. The channel devices will be used to implement the synchronization required to establish Shared Regions as well as in support of the UDS layer using shared memory transfer. Typically

---

[2] To resolve the ambiguity noted previously, these channels will be called **channel devices**.

this would be accomplished by a UDS process passing the address of a datagram (or fragment) through a virtual channel to a peer process on an adjacent machine.

Special interrupts are implemented in the Kernel to "poke" a neighbor when new information has been placed into the channel device; after reading the information the receiver then clears the channel device, allowing the sender to transmit any additional information.

A hybrid channel-packet has also been defined which causes the recipient to poke its neighbor. This can be used to support messages passed through shared memory from processor P_i to processor P_{i+2} (or P_{i-2}) with processor P_{i+1} (or P_{i-1}) being the recipient of the hybrid packet. This is required for correct operation of the algorithms controlling access to Shared Regions. A more thorough treatment of this topic can be found in section 4.1.2 of [Li87b].

CHAPTER III.

Communication Protocols.

As mentioned previously, one of the most important aspects in the design of any multiple processor system is the area of interprocessor and interprocess communications. Without a way for machines to communicate with each other there can be no cooperation and therefore no benefit in the interconnection. Similarly, if the method of communication is inefficient, it will hinder the performance of the overall system and, in the worst case, will slow down rather than speed up the execution of an application.

In order for interprocess communications to be efficient three basic factors must first be considered, both separately and in relation to each other. At the "lowest" level is the hardware portion of the communication system--the actual medium and controller hardware. Efficiency at this level depends not only on the raw speed of the medium but also on its reliability and on any special features offered by the controller (large packet size, error recovery and correction and so on). At tne other end is the actual algorithm used in the application. There are many ways to split a large problem into several smaller parts, each requiring different communication support. The choice of a distributed algorithm can be instrumental in achieving the desired efficiency level.

The last factor to be considered lies, logically, in between hardware and algorithms, and acts as a bridge or interface between the two. Like the hardware, it provides certain services, and like algorithms it enforces rules and guidelines for communications. This feature is commonly referred to as a Communications Protocol, and it can be as varied as the other two factors considered above. Many protocols and protocol families (described later) have been defined and many new ones are under investigation. Lately there has been much work done on tools to simplify protocol creation and verification. A somewhat slower development has been the thrust toward standardization. This chapter presents the concept of protocol hierarchies and describes one of the most common ones in existence--the International Standards Organization (ISO) Reference Model for Open Systems Interconnection (OSI). The notion of protocol families is defined and several existing protocols and protocol families are presented. To conclude the chapter, the requirements of the project in relation to communication protocols are discussed.

1. Protocol Hierarchies.

The task of transferring data between processes on different machines involves many intermediate steps. A multitude of problems have to be dealt with, such as error correction, fragmentation and routing, to name just a few. As in the case of operating systems, the designers of net-

working systems have found it appropriate to structure the design in a hierarchical fashion. The overall complexity of the task is thus distributed among several relatively simple steps. Functions are grouped together or separated, according to the service they provide. As in the layered operating system design of the previous chapter, each layer provides services to be used by the layer above.

In a hierarchical communications protocol the user's data are passed down between layers, each performing a certain function. After the data reaches the remote machine, this process is reversed: the data are passed up through the layers, each layer "undoing" what its peer had done on the sending machine. The end result is that the receiving process on the remote machine receives the same information that was sent by the sending process.

The same can be said about every level of the hierarchy. The information that is passed down from layer n on the sending machine is identical to the information sent up to layer n on the receiving side. Thus it is said that there is virtual communication going on between peer protocol levels. The lowest layer, which performs the actual transfer of data over the medium, communicates with its peer level directly, since there are no levels beneath it.

There are many ways of grouping the functions of a network into protocol layers. The only feature that is common to all is that the physical data transfer is done at the lowest level, while the interface to the user is done at

the highest. This is analogous to the hierarchical operating system structure, where the lowest layer is the hardware, and the highest layer implements the shell--or user interface.

2. The ISO/OSI Reference Model.

One of the best known communication protocol hierarchies is the ISO Reference Model for Open System Interconnections (OSI). This model (Figure 3) is composed of seven levels--from the Physical layer to the Application layer. Appendix B briefly outlines the levels in terms of the services for which each is responsible.

As with the layered operating system design each level implements services that are used by the levels above. Unlike the operating systems example, however, the OSI reference model is not a functional hierarchy. This means that the services of a lower level are available only to the layer directly above it. Once user data has entered some level it must go through every level down to the Physical layer--there can be no short cuts.

Each level implements a protocol by which it communicates with its peer level on the receiving side. In general these protocols imply the addition of some overhead to the data (in the form of headers and/or trailers) as they pass through each layer. On the receiving side this process is reversed, each layer stripping off the overhead pertaining

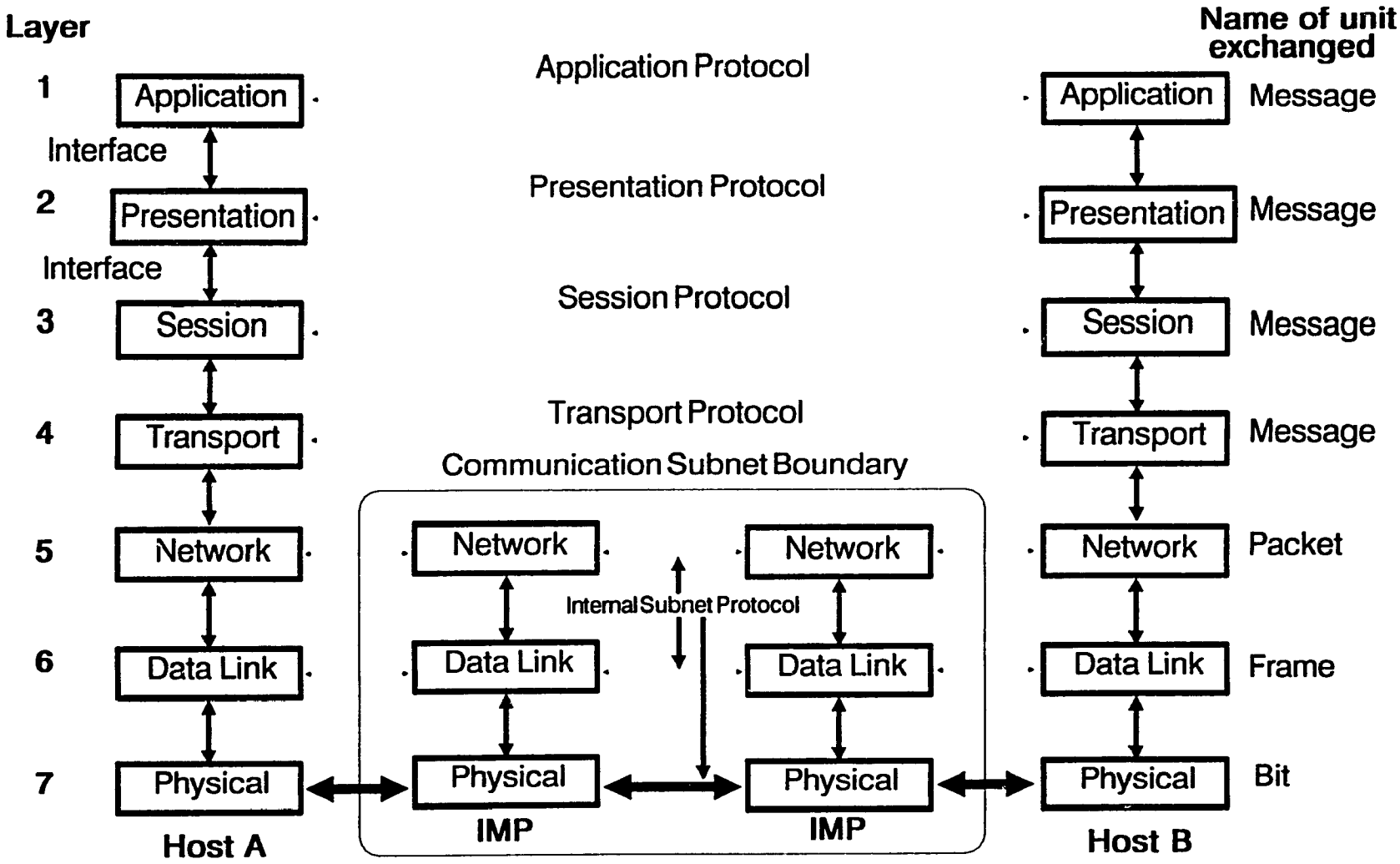

Figure 3. The ISO/OSI Reference Model

to that level only, before passing the data upward. In this fashion each level on the receiving end receives the same unit of data that was passed from its peer level at the sending side. Thus a virtual communication between peer levels is established, signified by the dotted lines in the diagram. Communication between the lowest levels is direct over the communication medium, and not virtual, which is shown by the solid line in Figure 3.

The lower three layers together make up the internetworking subsystem of the hierarchy. If a message has to cross a network boundary it will go through special network stations called "gateways". The box in between the lower three layers in Figure 3 represents crossing-over between networks through gateway nodes. The gateways only use the bottom three layers of the protocol hierarchy as they are only concerned with passing the message to the correct network.

The goal of the reference model is to subdivide communications into workable layers, so as to reduce the overall complexity of the problem. The division into layers follows five main principles as outlined in [Tane81]. One: a layer should be created where a different level of abstraction is needed. Two: each layer should perform a well-defined function. Three: the function of each layer should be chosen with an eye toward defining internationally standardized protocols. Four: the layer boundaries should be chosen to minimize the information flow across the inter-

faces. Five: the number of layers should be large enough that distinct functions need not be thrown together in the same layer out of necessity, and small enough that the architecture does not become unwieldy.

The reference model is just that--a reference. It does not correspond to any specific network implementation, nor do most network hierarchies map directly onto the model. In fact, the exact definition of each layer is clearly understood to be a "grey-area", which is exemplified by the fact that OSI has defined five classes of Transport Layer protocols [Pisc86]. The various classes, TP_0 through TP_4 provide the same service to the Session Layer, but each have different functionality, depending on the type of service provided by the Network Layer. Class 0 assumes a completely reliable Network service, and is not required to implement functions such as error recovery, flow control or out-of-sequence data, to name a few. Class 4, on the other hand, assumes that the underlying network is unreliable and implements all of the above (and other) functions. The various Transport Protocol classes are summarized in Figure 4.

The reason for this non-standard behaviour from a standards organization is the understanding that not all applications need the full services a network can provide. For example, speech or image applications do not need the extensive error correction required in file transfer protocols, so they should not be restricted by the time overhead

Function	Class				
	0	1	2	3	4
Error Recovery	NO	YES	NO	YES	YES
Expedited Data Transfer	NO	YES	YES	YES	YES
Explicit Flow Control	NO	NO	YES	YES	YES
Multiplexing	NO	NO	YES	YES	YES
Detection and Recovery from:					
Duplicated TPDUs	NO	NO	NO	NO	YES
Lost TPDUs	NO	NO	NO	NO	YES
Misordered TPDUs	NO	NO	NO	NO	YES
Corrupted TPDUs	NO	NO	NO	NO	YES

Figure 4. Comparing Transport Protocol Classes

incurred by this feature. An actual implementation may also combine the services of several layers into one protocol--this would only limit the application's access to the various levels. Also, some hardware allows for features of higher layers to be omitted. Taking shared memory as an example, correction and error-detection is not vital because of the inherent reliability of memory access. It can also be argued that given a clever memory allocation scheme, the shared memory-based communication will not require fragmentation and reassembly to be implemented.

As mentioned earlier, the ISO/OSI model is not the only existing network structure. Other network architectures include proprietary architectures such as IBM's SNA (Systems Network Architecture) and DEC's DECNET, as well as the ARPANET network and, more recently, the DoD architecture from the Department of Defense. The DoD architecture has seven layers, like the ISO model. The difference is in the Network layer, which DoD has split into Network and Internetwork sublayers, and the Presentation and Session layers which, in the DoD architecture, are combined into one level, called the Utility layer. Differences such as these are, in effect, cosmetic. They imply a slight variation in the division of communication functions into layers, rather than differences in the functions themselves. The approximate correspondence between the ISO/OSI layers and those of DoD and the other architectures mentioned above is summarized in Figure 5.

<table>
<tr><th>Layer</th><th>ISO</th><th>DoD</th><th>ARPANET</th><th>SNA</th><th>DECNET</th></tr>
<tr><td>7</td><td>Application</td><td>Application</td><td>User</td><td>End User</td><td rowspan="2">Application</td></tr>
<tr><td>6</td><td>Presentation</td><td rowspan="2">Utility</td><td>Telnet, FTP</td><td>NAU Services</td></tr>
<tr><td>5</td><td>Session</td><td>(None)</td><td>Data flow Control</td><td>(None)</td></tr>
<tr><td>4</td><td>Transport</td><td>Transport</td><td>Host-host</td><td rowspan="2">Transmission Control</td><td>Network Services</td></tr>
<tr><td rowspan="2">3</td><td rowspan="2">Network</td><td>Inter-network</td><td rowspan="2">Source to Destination IMP</td><td rowspan="2">Path Control</td><td rowspan="2">Transport</td></tr>
<tr><td>Network</td><td rowspan="2">IMP-IMP</td></tr>
<tr><td>2</td><td>Data Link</td><td>Data Link</td><td>Data Link Control</td><td>Data Link Control</td></tr>
<tr><td>1</td><td>Physical</td><td>Physical</td><td>Physical</td><td>Physical</td><td>Physical</td></tr>
</table>

Figure 5. Approximate Correspondence between the Various Network Hierarchies

3. Protocol Families.

As mentioned above, each layer in the reference model communicates with its peer through a protocol intended for that level. Each protocol presents an interface to the layer above it by which the services of that layer can be accessed. Unless the user application is prepared to access the network via the Physical layer, several protocol layers will have to be implemented in order to present the services of the network to the users. Several protocols, implementing various levels of the hierarchy, which are designed to interface with each other, are said to be a protocol family.

The concept of a protocol family is needed because the OSI reference model, or any such model, does not enforce any restrictions on the design of protocols. Even if a protocol maps directly onto one (or more) of the model's levels, that does not imply that another protocol which maps onto the next level of the hierarchy will be able to interface with it correctly. This is because the protocols specify only the peer-to-peer interactions, and leave the interface between layers as an implementation-defined detail.

Several protocol families exist. TCP/IP is a protocol family used in the US Internet which implements the services of the Transport and Network layers. TCP (Transport Control Protocol) implements the services of the Transport layer and interfaces with IP (Internetworking Protocol)--which implements the services of the Network layer.

The IEEE 802 family of protocols comprises the 802.2 Link level protocol and several Physical level protocols (802.3, 802.4, 802.5). The specifications ensure that the Link-level protocol is compatible with any of the Physical-level protocols.

The X.25 recommendation specifies protocols for the bottom three layers of the ISO model. It can be used to support TCP/IP, which means that the Network layer's functions are distributed between IP and the top protocol of X.25. In the DoD framework these would map onto the Internetwork and the Network layers respectively.

4. Project Requirements.

Several current multiprocessors and network architectures were examined in terms of the communication subsystems of the operating systems implemented. These included the Sprite project [Welc86, Oust87], the Newcastle Connection [Brow82, Panz85], the Cm* and Cmmp architectures from Carnegie-Mellon university and the operating systems designed for them (StarOS [Jone79], Medusa [Oust80], Hydra [Wulf74]), Roscoe [Solo79], Locus [Pope81] and V [Cher84, Cher88]. Some were clearly networks rather than multiprocessors, (Newcastle Connection, Roscoe, Cm*), and were studied solely for the protocols employed and to understand how these protocols were chosen. Others were studied closely because of the hardware similarities to the HM architecture (Locus, V).

This research provided an insight into the communication requirements of distributed systems and multiprocessors and showed some of the common problems encountered during the design and implementation of such systems. In particular it was found that shared memory was not as common as networking. If shared memory was supported it was usually in the form of one large memory accessed by all processors through a cross-bar switch as in Cmmp. In Cm* all memory is also visible, but each processor has its own local memory and remote memory is accessed through the network rather than directly. In all instances shared memory was viewed solely as a resource to be accessed directly, and not as a medium for the physical layer of a communications system. (In Li's design, Shared Regions are available to processes to be used as needed, but are also used by the communications subsystem to transfer packets between neighboring stations.) Remote procedure calls were found to be used extensively (Newcastle Connection, Sprite, V), as were message-based communications (Roscoe, StarOS). Generally it was seen that remote procedure calls and higher level protocols, such as pipes, could be supported by message-based communications such as are provided by the datagram protocol in [Panz85].

The Homogeneous Multiprocessor is intended to be used as a general-purpose multiprocessor. According to Chanson et al., [Chan84], LAN messages fall into three main categories: Remote service requests/replies, System generated

messages and Stream-type messages. The first maps directly onto Remote Procedure Calls and is supported by that layer. The second type refers, in general, to intra-protocol control messages (these protocols can be at various levels in the ISO reference model). This type of message can usually be supported by the protocol below the one which is sending the control messages. For example, the RPC layer will most likely use the UDS to send protocol-specific control messages to remote peers. The third type of message is a pipe; it is provided by the Communications layer of the HM-Nucleus.

To the above three message types we add another--interprocess messages. These are similar to the System-generated messages in that they do not require any connection orientation. As such they would be implemented less efficiently using the pipe-based communications provided by the HM-Nucleus. Using the RPC layer, on the other hand, would restrict the use of such messages to conform to the RPC protocol. As an example of this type of message traffic consider a master-slave algorithm where a master process sends some data to be processed by a slave process, and then waits for the response. This exchange is similar to the request/reply semantics of the RPC layer, but it would be unreasonable to force the implementation of the slave process to be a general-purpose server. A connection-oriented scheme would be similarly undesirable, as the connection

set-up overhead and complexity could not be justified in such a simple exchange.

What makes this type of message different from the System-generated messages described by Chanson is that the Communications system must provide a user-level interface to these services, whereas a protocol layer is already provided (the UDS layer) with this interface. This means that the Communications Subsystem of the HM-Nucleus must be flexible enough to provide to the user an interface to virtually every level in the ISO/OSI model.

CHAPTER IV.

Design of the Communications Subsystem.

Up to this point, the hardware features of the Homogeneous Multiprocessor have been introduced and Li's proposed structure for the HM-Nucleus has been given.

In this chapter, the requirements imposed by the projected uses will be reviewed, and a design developed for the Communications Subsystem.

1. Communication Protocols.

The communications level proposed by Brown, Denning and Tichy [Brow84] supports a single interprocess communication (IPC) model: pipes. This was motivated by their desire to propose a system in which the only "construction" primitives visible to the user applications were processes and pipes [Brow85]. Details of the mechanisms used to achieve this communication were suppressed in their exposition; other levels are clearly necessary to achieve the actual communication, but inclusion of these layers in their model would have caused it to be unduly complicated [Brow85].

In the design for the HM-Nucleus [Li87a, Li87b], the Communications Subsystem consists of three layers: Universal Datagram Services (UDS), Remote Procedure Call (RPC), and Communications (COM).

The COM layer is assigned the responsibility for pipes (implemented through the H-Network), and also for (user)

channels and a remote processor signalling mechanism. In order to achieve high throughput, we have added the ability to move (pipe) data through the shared memory between two processors to the list of responsibilities of this layer.

The addition of an RPC layer to Brown's model reflects the expectation that the remote procedure call will be a most useful paradigm for the proposed applications on the Homogeneous Multiprocessor.

The proposal for a UDS layer reflects the fact that many services (Pipe management, Shared Region management, Global Memory management, remote procedure calls, and, later on, distributed files and distributed directories) will require a datagram-oriented service for their convenient implementation. The UDS layer hides the details of routing (process location) and fragmentation.

The COM layer provides connection-oriented services. However, use of a connection-oriented protocol for lower levels would degrade the performance of the UDS and RPC layers, where connection-orientation is not necessary. It is more efficient to include connection schemes only where they are needed, and refrain from implementing them at lower levels.

The UDS layer could be built on top of the hardware driver and provide services of the Data Link as well as the Transport levels. However, since the HM architecture supports two communication media, and because the UDS layer must provide uniform access to these media (and more as they

appear), it is simpler to implement the UDS datagram service on top of a Link-level protocol which, in turn, provides uniform media access. This reduces the functions of the UDS layer to only the fragmentation and reassembly of datagrams (since the packet size at the Link level is generally limited by the medium controller), and routing. It is of course possible to handle fragmentation at the Link level or support transparent access to various media at the UDS layer, but this would make one of the layers unnecessarily complex.

The protocols implemented then, need to be flexible enough to provide both connection-oriented and connectionless services to the user. Barring the use of a general-purpose protocol that supports all types of services, the above stipulation would require that the Communication Subsystem provide user interfaces to several levels of the ISO/OSI hierarchy. In addition, the Link level protocol chosen must be flexible enough to provide a uniform access to the H-network, the shared memory and any other media which may be connected to the architecture in the future.

In view of the above requirements, the merits of the protocol families mentioned in Chapter III will be discussed.

TCP/IP.

TCP/IP is widely used in networking and is also available for a Unix environment with Ethernet support. However it was designed with Long Haul Networks (LHN's) in mind and

contains many features unnecessary in LAN implementations (e.g., internet addressing, two-level checksumming, byte-level sequencing). These features produce considerable overhead and reduce the efficiency of a LAN [Chan84].

LNTP.

LNTP provides the services of TCP/IP which are relevant to LANs [Chan84, Chan85]. It removes the unneeded features of TCP (see above) and stresses, instead, the features required for efficient LAN communications (avoiding fragmentation, simple flow control, selective retransmission). Unfortunately, LNTP is not a standard protocol, nor is it likely to become one. Also, it implements connections, which in our scheme have been relegated to a higher level.

IEEE 802 family.

The IEEE 802 family of protocols was chosen for several reasons. First--it is a recognized standard, and as the push toward standardization continues will become more widely accepted. Second--it includes specifications for various Physical level protocols (802.3, 802.4, 802.5) to support different communication media. The Link level protocol (802.2) is specified in such a way that it can interface with the various Physical level protocols. The standard for CSMA/CD (802.3) can be adapted easily to suit the CSMA/CF scheme used for the H-Network. As far as shared memory is concerned, it would be a relatively simple task to

define a protocol which would interface with the 802.2 Link level protocol (see chapter III). Finally, this protocol specifies two modes of operation. LLC Type I supports connectionless services using a very simple and efficient protocol. LLC Type II provides both connection-oriented and connectionless services and is compatible with LLC Type I. The simplicity of LLC Type I is very attractive for an initial implementation, while LLC Type II can be added later if desired, without changing the rest of the Communication system.

2. Relationship to the Memory Management Subsystem.

There is an interaction between the Memory Management Subsystem and the Communications Subsystem which must also be explored. In [Li87a], the Memory Management Subsystem proposed for the HM-Nucleus consists of two layers: Physical Memory Management (PMM), and Virtual Memory Management (VMM).

Li assigns to the PMM the responsibility for the allocation and deallocation of memory space for processes and communication packets, in addition to the implementation of virtual-to-physical memory mapping. The allocation is done by **AssignSegments**, using the Buddy algorithm. In order to conserve descriptors in the Memory Management Unit, the minimum size of an allocation is 4K bytes. A virtual-to-physical mapping is bound using **BindSegment**. Both of these routines have a (user) task identifier as one of their

parameters. Another function which is assigned to the PMM is garbage collection of segments that are no longer needed (either the owning process has been destroyed, or the communications packet has been consumed).

The VMM is responsible for assigning and managing virtual space for processes: program, data, and stack space for user programs; Shared Regions; and Global Memory.

Shared Regions are allocated by the Virtual Memory Manager (VMM) in blocks of at least four kilobytes and can be specified to be either guarded or unguarded (the difference between the two will be described later). Access is effected through the **CreateRegion**, **BindRegion**, **EnterRegion** and **ExitRegion** calls of the VMM. The first two calls, respectively, define a Shared Region and give it a name to be used in later calls. Processes on adjacent machines subsequently acquire the capability for the region by issuing the **GetName** call, implemented in the Table layer. The **EnterRegion** and **ExitRegion** calls implement the mutual-exclusion algorithm needed to ensure safe sharing of data by up to three processors. The algorithm, described in [Li87a], uses one central semaphore, in the local memory of the owner of the Shared Region, and three spin-locks, in the local memories of each of the three sharing processors. The spin-locks are used to control access to the semaphore which, in turn, ensures controlled access to the shared data. With this arrangement busy-waiting performed in one of the sharers will not need to access the interbus switch, which

would interfere with the operation of the owner processor. The mutual exclusion is needed to implement guarded regions; if the region is created unguarded, the **EnterRegion** and **ExitRegion** calls are transparent. This implementation allows applications to make a compromise between security and efficiency, according to their individual needs.

The synchronization required among the spin-locks and the control semaphore is implemented using the hybrid channel packets described in Chapter II.

The Global Memory is implemented by replicating the data in the memory of each processor, and then using the communications subsystem to control updates. Three update algorithms are proposed in [Li87a], which make use of the particular properties of the H-Network and shared memory for communication.

The RPC layer builds on the UDS layer, and is independent of the COM layer. It does not interact with the memory management software, and therefore will not be discussed further in this section.

Four points concerning the interaction among the UDS, COM, PMM and VMM layers can now be raised. The first three relate to the management of the memory used by the Communications Subsystem, and the fourth addresses the visibility of channels. To avoid ambiguity in what follows, we define **above** to mean "further away from the hardware".

Li suggests that messages that are going to be exchanged between adjacent processors be sent via the shared

memory, and that this memory be allocated as a Shared Region. If so, then the shared memory "driver" for the physical level in the UDS must be above the VMM. However, the VMM utilizes channels to achieve coordination among the users of a Shared Region, and these channels are assigned as part of the responsibility of the COM layer. The COM layer is two levels above the driver. (The driver is located inside the UDS layer.) This represents an unfortunate circularity. Therefore, the Communications Subsystem cannot use Shared Regions for its buffers.

The packet size adopted for common memory transfers is likely to be significantly smaller than 4K, which is the minimum allocation size for the PMM. Thus, allowing communications packets to be garbage collected by the PMM does not seem appropriate. Therefore, the shared memory driver will have to allocate a (relatively) large buffer, and then manage it by itself.

As noted, one of the parameters of the **AssignSegments** and **BindSegment** calls in the PMM is a task identifier. As the Communications Subsystem is acting on behalf of all processes, there is unlikely to be a user task identifier to associate with a shared memory segment that has been assigned for communication use. Therefore, the task identifier must be dropped as a parameter for the PMM primitives.

User channels represent a medium with significantly different properties from those of the H-Net. From the point of view of a user process, it may be immaterial which

medium is used. However, from the point of view of the software in the VMM, it is essential that certain features of channels be visible to it, and that the identity of the specific processor being addressed via the channel be known. It is therefore necessary to both expose the detailed features of channels (within the HM-Nucleus), and hide those features from user processes (outside the HM-Nucleus).

3. Structure of the Communications Subsystem

The structure chosen for the Communications Subsystem must satisfy the following requirements:

a. Provide a device driver for the H-Network.

b. Provide a device driver for nearest-neighbor communication, using the shared memory and the channel devices.

c. Provide separate, low-overhead access to the channel devices, for use by the special algorithms within the HM-Nucleus (such as those that support Shared Regions and Global Memory).

d. Integrate the H-Network and the shared memory communications paths into a uniform, IEEE 802.2-based subsystem, with provision for adding other media at a later date.

e. Provide a simple mechanism for the construction of modules to implement the UDS, RPC, and COM functionalities.

f. Ensure user access to any level of protocol within the Communications Subsystem, preferably with a uniform

interface. "User" in this context may mean: i) an HM-Nucleus entity implementing Shared Regions or Global Memory; ii) a user application built on the present HM-Nucleus, executing in supervisor mode; or iii) a user application on a future HM-Nucleus, executing in user mode.

g. Manage its own memory, including garbage collection, using a buffer pool requested from the Physical Memory Manager at system initialization time.

The chosen vehicle for this flexibility is the STREAMS facility, which is introduced in the next chapter. The next two subsections outline the design of the channel devices and their use in providing a nearest-neighbor communications path. These will provide device drivers for use within the STREAMS framework.

4. Channel Devices

As has been mentioned previously, protocols for shared memory-based communication are not as abundant as those for more conventional media. As a result, it is impossible to select a "standard" or to adapt an existing protocol to suit the requirements of the project. This section outlines a proposed protocol for channel devices, and the next section outlines how channel devices can be used to provide shared memory-based interprocess communication in the Homogeneous Multiprocessor.

Channel devices provide a uniform mechanism to support several algorithms used in higher-level modules. As proposed by Li [Li87a], channels provide a small buffer, and a mechanism for interrupting a neighboring processor. Li also suggests that virtual channels be provided, as there are several proposed algorithms in the HM-Nucleus that rely on the efficiency of direct transfers between adjacent processors for their operation. Finally, Li proposes a remote signalling mechanism, using a special "hybrid" packet in a channel device, to permit the interruption of processor Pi-2 or Pi+2 from processor Pi.

Channels are not efficient for direct use in interprocess communication. The primary reason for this is the fact that the size of the data part of a channel is (necessarily) small. The second reason is that only one channel is defined for each (left and right) neighbor. Additional channels would require the **poke** operation--which is, essentially, an interrupt--to be capable of passing the channel address. In Li's design, the addresses of right and left channels are fixed at compile time and constant across all nodes of the multiprocessor. The third reason is that since only one channel is defined for each neighbor, the response to a **poke** and consumption of the channel information must be completed in a short time. The first is an interrupt service, and as such poses no problems. The consumption of the data, however, may depend on a user process, and cannot be guaranteed to be prompt. At best, the data can be copied

into a local buffer, thus freeing the channel quickly. That involves an extra copy, however, which should be avoided if possible.

Our proposal differs slightly, in that it distinguishes between the signalling mechanisms and the management of the associated buffers. A channel device consists of a data structure containing three elements: a channel number, a data value, and a memory address. Separate channel devices are provided for each direction of data transfer; these are assigned fixed addresses which are the same for all processors. The value zero in the channel number field is used to indicate that the contents of the channel have been consumed. The high order bit of the channel number is used to indicate direction. Certain channel numbers are assigned to indicate special functions (such as remote signalling). The data value is used when the special function to be performed involves setting a byte somewhere to that value. The memory address gives the location of a buffer being transferred, or the address to which the special function is to be applied. The memory associated with a buffer of data is managed by the sender, and not by the channel device driver.

The proposed operation is as follows:

1) The sending channel device driver copies the channel data into the data structure.
2) The **poke** operation is used to interrupt the adjacent processor.

3) The interrupt handler that responds to the **poke** operation examines the channel. If the channel number is one which has been allocated to a special function, then that function is performed. Otherwise, the contents of the channel are enqueued for use by upper-level software.

4) The interrupt handler then sets the value of the channel number field to zero.

The delay required for synchronization of the two processors can be achieved in one of two ways: either the sending channel device driver can loop, examining the channel number field until it becomes zero, after it has **poked** its neighbor, or it can test for the zero value prior to filling the channel in the first place. The first approach ensures that the receiver has acted, prior to letting the sender continue. The second approach is likely to result in less contention between the processors, as the probability of the channel being full at the time of access by the sender should be quite small.

In order to permit the greatest flexibility, the entry point address of the special functions is definable using an **ioctl** call to the driver. This will provide a user-specifiable link between a channel number and the function which is executed when it occurs.

5. A Proposed Shared Memory Protocol.

We use the channel device only to pass the address of a data packet. The packet itself is stored in the (left or right) shared memory partition of the sending processor. The shared memory segment from which buffers for packets are obtained is allocated by the PMM. However, since the PMM allocates segments in 4K byte increments, the Communication Subsystem must maintain its own buffer pool. This includes any allocation, deallocation and garbage collection.

When a buffer is to be sent to a neighbor processor, its address is placed into the corresponding channel device and the neighbor is poked. The receiving processor, knowing which neighbor performed the poke, can easily map the received address to the corresponding shared memory partition. This address can then be queued to be processed at a convenient time by the communication layer. The channel device is immediately marked as empty and can be used for further communication.

Since the buffer will not necessarily be processed immediately, it must carry a flag specifying to which processor, the neighbor or the owner, it currently belongs. The flag is set to "neighbor" when a packet is copied into the buffer, and it is reset to "owner" when the packet data has been consumed.

Since no explicit acknowledgement is sent to the packet originator, specifying that the packet has been received and processed, the buffer manager for the Communication Subsys-

tem must perform garbage collection. Initially the buffers are placed in a linked list. When this list becomes empty, the allocation routine invokes the garbage collector. The garbage collector finds all consumed buffers and builds a new linked list. If no consumed buffers are found, the garbage collector will alternate between sleeping and collecting, until the linked list of buffers is no longer empty. Alternatively, the garbage collector can request another shared memory segment from the PMM, and form an additional buffer pool out of this block.

The above scheme overcomes limitations of the channels, as described previously. In addition, the size of the buffers does not have to be fixed. Various quantities of different buffer sizes can be allocated, depending on the demands and nature of communication. Also, since packet consumption is no longer a time-critical operation, the packet can either remain in the original buffer until delivery to the user, or be copied to an intermediate buffer, as is appropriate.

This protocol can easily be incorporated into STREAMS. There is already a buffer manager in STREAMS, very similar to the one described above. It will be a simple task to expand that manager to maintain shared memory buffers, or to add a separate manager to handle this task.

CHAPTER V.

Unix STREAMS.

This chapter describes STREAMS--a facility designed for the UNIX operating system to be used as a major building block in providing networking support for Release 3.0 of system V [AT&T87a, AT&T87b]. It is a communication subsystem which is compatible with the current UNIX character I/O interface, yet flexible enough to allow implementation of various hierarchical protocols used in network architectures. STREAMS does not implement any such protocols, but rather provides developers with an environment and a set of working tools that reduce protocol implementation to a relatively simple task.

1. Overview.

Central to STREAMS is the concept of a **Stream.** Basically, a Stream is a bi-directional data path between a user, in user space, and a communication device driver, in kernel space. Buffer allocation, scheduling and flow control are all incorporated into STREAMS, which greatly reduces the developer's task. Another feature of STREAMS is **polling,** which allows a user process to monitor several separate Streams and receive input asynchronously.

A Stream is a chain of kernel-resident STREAMS modules, which can be linked together or taken apart dynamically. A module encompasses a set of (usually) reentrant procedures

which perform some pre-specified action on data passing between the user and driver. The topmost module of a Stream is called a **stream-head.** This is a standard STREAMS module which provides the user interface to the subsystem. Other than packaging the user-space data into kernel-space messages (and vice-versa), no processing of data is performed by the stream-head. The interface is provided by the usual UNIX read/write/ioctl system calls as well as by two new calls--**putmsg** and **getmsg.** These two calls are analogous to write and read, but allow protocol-specific information to be passed along with the data. They also preserve message boundaries.

The last module in a Stream is a **device driver** module. This module implements the interface between the kernel and the communication hardware. The procedures in a driver module are not reentrant (the reasons for this are too involved to be discussed here), so a separate instance of a driver module must exist for every hardware interface, while other modules can be used in several Streams without duplicating their code.

A minimal Stream, consisting of a stream-head and a driver, is created when the device driver module is opened. The open system call recognizes that the device is a STREAMS device and calls the appropriate STREAMS open routine. It is during this time that the non-shareable, dynamic part of the head and driver are allocated. Following this, other kernel-resident modules can be pushed and popped below the

stream-head in a LIFO manner. Thus, if several modules, each implementing a certain layer of the OSI hierarchy, exist, the user can create Streams that will provide protocol services at various levels in the hierarchy. Also, since a standard stream-head is always the top module in any Stream, the user is provided with a consistent interface, namely the **getmsg** and **putmsg** calls. Figure 6 illustrates this principle by showing two Streams providing a Transport-level service and a Link-level service.

2. STREAMS Modules.

A module is a collection of procedures designed to process messages passed in a Stream. Due to the bidirectional nature of a Stream, the module has a **downstream** and an **upstream** part, corresponding to message flow from the head to the driver and from the driver to the head, respectively. Two types of procedures are used in a module to process messages. The **put** procedure is called from a preceding module to propagate a message along the Stream. The processing of data by the **put** procedure is performed immediately. The developer may chose to place some non-time critical processing in the **service** procedure. Typically, the **put** procedure will perform some decision or management functions, then enqueue the message to be processed at a convenient time by the **service** procedure. The STREAMS scheduler will execute the **service** procedure automatically at a later time. Each module must have both a read-side and

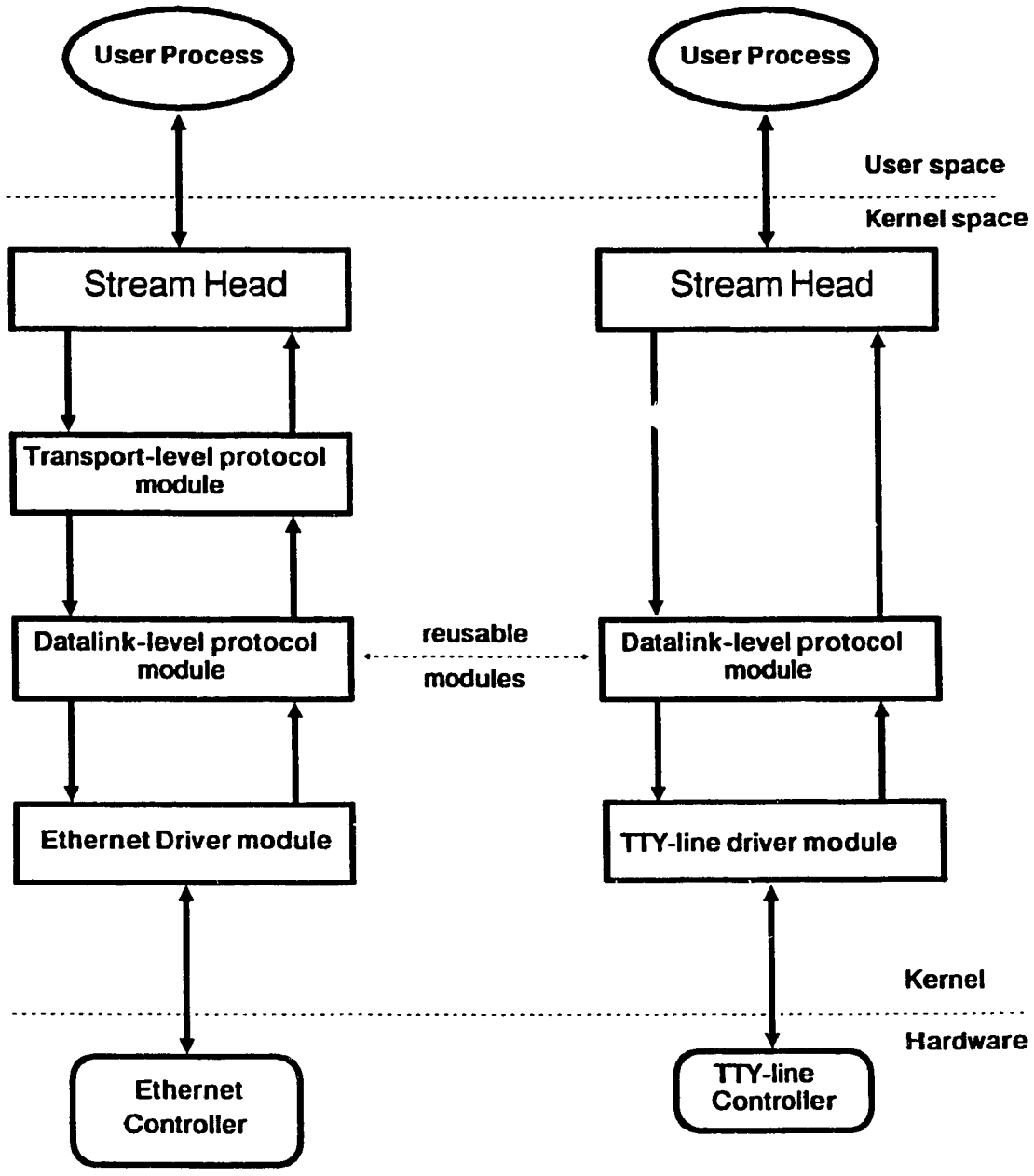

Figure 6. STREAMS Modularity:

Stream (a) offers a Transport-level protocol service, while Stream (b) offers a Datalink-level servcice.

a **write-side put** procedure, but **service** procedures are optional.

Each side of the module is defined by a **qinit** structure (see Appendix C) which contains pointers to the corresponding **put** and **service** procedures, among other things. The overall module is represented by a **streamtab** (see Appendix C) structure, which contains pointers to the read and write-side **qinit** records of the module (**rinit** and **winit**). The **streamtab** structure also contains pointers to two other **qinit** structures (**muxrinit** and **muxwinit**) which are used only by multiplexing drivers (described in section 9). Two other structures are used to describe a STREAMS module. These are the **module_info** and **module_stat** structures. Briefly, the first contains information about the module, such as its name and ID, minimum and maximum packet sizes, and high and low-water marks for flow control purposes. The second structure, which is optional, contains statistics, such as the number of invocations of the module's procedures, the number of times flow control has been applied, and so on. Pointers to these two structures are placed in the **qinit** records for each side of the module. Pointers to the **streamtab** structures for driver modules are placed in the **cdevsw** table and used in the open routine to initialize the dynamic parts of the driver. A similar table, **fmodsw**, is used for ordinary STREAMS modules.

3. Kernel Data Structures.

Two data structures are fundamental in STREAMS. They are the structures which define the STREAMS messages and Queues. Messages allow data to be passed within STREAMS without the need of copying, even if header or trailer information must be added to the original data. Queues are the dynamic parts of modules which allow multiple instances of a module to exist in STREAMS.

Messages.

STREAMS is a message-based system, in that all information transfer within a Stream is carried out by passing STREAMS messages between linked modules. A STREAMS message (see Appendix C) is a linked triple consisting of two control blocks and one data buffer. A message on a Stream may consist of several such triples linked together.

The first control block, **dblk_t**, is specific to the data buffer. It describes the location and size of the buffer, as well as the type of the message to which the buffer belongs (STREAMS defines 18 message types, listed in Appendix D). It also contains a reference count (explained below). The second control block, **mblk_t**, defines the relationship of the data buffer to the rest of the message, and sometimes to other messages in a Stream. It contains a pointer to the next block of the message, as well as two pointers to the previous and next message, which allow queues of messages to be maintained. The reference count in

the **dblk_t** structure permits several **mblk_t** records to reference the same data buffer, allowing replication of read-only data without physical copying. There are also pointers to the next data byte to be read and written, allowing various messages to refer to different parts of the same data buffer.

Queues.

The second prominent data structure used by STREAMS is the Queue. Essentially, a Queue is one half of the dynamic portion of a STREAMS module. A Queue is defined by the **queue_t** structure (see Appendix C). A pair of these is allocated whenever a STREAMS driver is opened or a module is pushed onto a Stream. Several fields are initialized when a **queue_t** record is allocated. These include a pointer to a **qinit** structure and the parameters from the **module_info** record. The parameters are copied because the **module_info** is a read-only structure. The copies in the **queue_t** record can be altered to tune-up the performance.

The **queue_t** structure also includes pointers to the next Queue in a Stream and to the next Queue on the STREAMS scheduler queue (described in a later section). Two other fields point to the first and last message enqueued on the Queue. This allows Queues to maintain a doubly-linked list of messages, waiting to be processed by the **service** procedure. The last field in the **queue_t** structure is a pointer to a private data structure. It is developer-dependent and

is used to allow a module's procedures to be reentrant, while still allowing private data structures in each instantiation of a module.

4. Buffer Management.

STREAMS maintains its own buffer pool from which it allocates data buffers for messages. At the stream-head interface, user data (which is in user space) are copied into these buffers (which are in kernel space), and remain there until the message reaches the driver. Only message pointers are passed, so no additional copying is necessary.

Buffers can be allocated in various sizes and at three priority levels. When the buffer pool becomes depleted, a low-priority request (such as a user **write** or **putmsg**) will be denied in favour of a high-priority request (such as input on the driver side). Utility procedures exist to allocate and free buffer space, as well as to test for buffer availability (a list of utilities is given in Appendix E). A routine (**buffcall**) is also included to assist developers in recovering from buffer-allocation failure.

5. Scheduling.

After the initial interface call, data are passed along a Stream without additional user intervention. When a **put** procedure in one of the Stream's modules enqueues the message, the original user call returns, and the message becomes the responsibility of the STREAMS facility. In

order for the message to propagate any further along the Stream, the module's **service** procedure must be invoked. STREAMS performs this scheduling task independently from the UNIX kernel's scheduling routines.

The typical scenario is as follows. The user issues a **write** or **putmsg** call, which creates a STREAMS message and invokes the write-side **put** procedure in the top-most Stream module. If all the processing is performed by the **put** procedure, its last action is to invoke the **put** procedure in the next module in the Stream. This will continue until the message reaches either the driver or a module with a **service** procedure. In the case of the **service** procedure, the module's **put** procedure enqueues the message and returns, eventually completing the initial user call. Since the user is now free to continue, it is the responsibility of the STREAMS facility to ensure that the **service** procedure is eventually called.

The STREAMS scheduler maintains a list of all Queues which contain messages to be processed by the **service** procedure. For each of these Queues, the scheduler locates the **service** procedure (from the **qinit** pointer in the **queue_t** structure) and invokes it. This action is performed automatically, without any user intervention.

6. Flow Control.

It was mentioned previously, that STREAMS implements flow control to aid in the development of modules and

drivers. Flow control is used to prevent fast devices from swamping slower ones. In STREAMS it applies more to modules than to devices, except in the case of driver modules.

The pair of Queues associated with each module keep a count of all data bytes in their message queues. This is a weighed count, where the bytes in short message buffers will have a greater or lower weight than bytes in large buffers, according to system requirements. In addition to the byte count, the Queues also maintain a high and low water mark, to determine when flow control should be enabled or relaxed.

Every time a message is added to the message queue, the **putq** utility automatically increments the byte count. If the new byte count exceeds the high water mark for the Queue, the state variable is changed to include a flag which specifies the Queue as being full. Whenever a message is removed, the **getq** utility automatically decrements the byte count. If the Queue had been full and the message removal brings the byte count below the low water mark, the full-flag is removed and back-enabling is performed. Back enabling is defined later.

The following guidelines should be observed to ensure that flow control works properly. Messages should be placed on queues using the provided utilities--**putq** and **putbq**. Messages should only be removed from queues using the **getq** utility. These utilities ensure that the byte count is updated correctly, manipulate the state flags and perform back-enabling.

Priority messages are not subject to flow control and are processed immediately. For ordinary messages, however, he following guidelines should be observed.

After the **service** procedure removes a message from the queue (using **getq**), but before any processing is performed, the **canput** utility should be consulted to determine whether the forward path along the Stream is blocked due to flow control. The **canput** procedure searches along the Stream for the next Queue with a **service** procedure (if such a Queue exists), and checks whether that Queue's state variable contains the full-flag. If **canput** fails, the calling Queue becomes blocked (indicated by a state-flag) and the message is returned via the **putbq** utility. A blocked queue can still accept messages, but, being unable to process them, will eventually become full. In this way, flow control will travel up or downstream until it reaches the driver or the Stream head.

When the low water mark is reached as a consequence of removing a message from the queue, back-enabling is performed. The **getq** utility searches in the opposite direction along a Stream, trying to locate any Queue which is blocked as a result of flow control. If such a Queue is found, it is enabled via the **qenable** utility, which removes the blocked-flag and submits the Queue for scheduling. If more than one Queue had been blocked, the first re-enabled Queue, after processing enough messages, will also fall below the

low water mark. This chain-reaction will ensure that all blocked Queues will eventually be back-enabled.

Flow control only applies to modules with **service** procedures, which should be evident from above. Without **service** procedures no messages ever get enqueued, so no accumulation of unprocessed messages would occur. Flow control can be regulated by adjusting the high and low water marks of the Queue. Since flow control and back-enabling propagate along connected STREAMS modules, it will not be a substitute for end-to-end flow control of transport-level protocols, nor will it cross over pseudo drivers in multiplexed Streams configurations (multiplexors are discussed in a later section).

7. Polling.

The **poll** system call allows a user process to synchronously monitor the input and output of several Streams simultaneously. The **read**, **write**, **putmsg** and **getmsg** calls support synchronous I/O over one Stream, where the user process will be suspended until data is available to read or space is available to write. The **poll** system call, in conjunction with the STREAMS signalling facility, allows a user process to perform asynchronous I/O on one or more Streams.

Polling works as follows. For each Stream to be polled, the user specifies the Stream file descriptor and the events to be reported (these events include arrival of

input data, arrival of a priority message and relaxation of output flow control on the Stream). This information is put into an array of records, which also contain a field for specifying returned events (these include fatal error, hangup condition, invalid file descriptor or no event). This array, along with the number of Streams to be polled and a timeout period are passed as parameters to **poll**. On return, the user checks each returned event and performs whatever action is required.

The **poll** system call allows a user process to synchronously monitor several Streams. The signalling facility of STREAMS, used with polling, allows a user process to monitor several Streams asynchronously.

An **ioctl** call specifying the I_SETSIG command is used to request STREAMS to send a signal to the user process when a specific event has occurred. These events include the three polling events (input, priority input and output), and notification of the arrival of a special message, containing a signal from a downstream module or driver.

The signal catching process can then use **poll** to determine which Stream has caused the signal, and process the corresponding event.

8. Utilities.

Many utility tools are included in STREAMS to facilitate the task of developing modules. Appendix E contains a list of these utilities along with a brief description of

each. Included are many macros which perform often-used tasks needed to move messages along a Stream (**putnext, getq, qreply**). Others provide some (hopefully) common message-processing routines (**linkb, adjmsg, pullupmsg**).

It is strongly suggested that a developer use these utilities whenever possible, in place of writing additional code. One reason for this is the saving in code space. Another is that some of the utility routines have certain side-effects which are essential to the proper operation of a Stream. For example, the purpose of the **putq** routine is to place a message on a message-queue, but a rather significant side-effect is that the associated Queue is placed on the scheduler queue to be serviced.

9. Multiplexing.

As was described previously, a Stream is a data path between a user process and a driver. In most applications a linear connection of various STREAMS modules is sufficient. Other applications, however, require the ability to multiplex several Streams in a variety of configurations. One such application is a terminal windows facility, where information coming from one physical line must be directed among several windows. A similar configuration is needed for any communications protocol which supports multiple users of the services it provides. A different configuration would be needed by a Network Layer protocol, which must

route data over several lower modules, each supporting a different communication medium.

STREAMS supports the above configurations with its notion of a **multiplexing pseudo-driver.** It is called a pseudo-driver because like a STREAMS driver, it must initially be opened as a Stream. In its final configuration, however, the multiplexor will not be the last module in the Stream--hence it is a **pseudo**-driver.

The first configuration, as in the windows example, is an upper multiplexor, and it requires no special features on the part of STREAMS. In order to have several Streams connected above requires only that each is opened with a different minor device number accompanying the major device number which specifies the pseudo-driver. Of course, the code in the driver's **open** procedure must be capable of supporting minor device numbers and setting up the necessary information, so that the processing procedures can correctly route data among the different upper Streams.

A lower multiplexor, as demonstrated in the Network protocol example, is more complicated. The lower sections in such a configuration must each be opened as a separate Stream, and then linked underneath the multiplexor. This linkage is performed by issuing an special **ioctl** system call on the multiplexor Stream, naming the lower Stream to be linked. The system call results in a specific message being sent down to the driver. The write-side **put** procedure in the driver processes the message and performs the necessary

steps, which include saving the address of the linked Stream in a global structure, which is accessible by all of the driver's procedures.

A multiplexor consists of two parts: the upper part, and the lower, linked, part. This duality accounts for the two special fields in the **streamtab** structure (**muxrinit** and **muxwinit**). The lower part contains procedures, but is not allocated any **queue_t** records. Instead, when a lower Stream is linked under the multiplexor, the **queue_t** structure of that Stream is modified to point to the multiplexor's lower procedures. This type of linkage means that the upper and lower Streams are not physically connected. For this reason STREAMS flow-control cannot propagate across a multiplexor, and the driver procedures must handle flow-control internally. In a typical implementation, when STREAMS flow-control restricts the passage of messages above or below the multiplexor, the corresponding messages are simply discarded.

To hide the complexity of multiplexed Streams, these configurations are usually set up by a daemon process. This process opens all the necessary Streams and performs all the linking. All that a user process is required to do to access the configuration is to open a Stream to the topmost multiplexor, specifying a new minor device number. Usually, the minor device number zero is reserved for the daemon, which becomes the controlling process for the Stream configuration. The multiplexor procedures will typically check

that special control messages, such as link and unlink messages, are sent only over the control Stream.

Multiplexors allow Streams configurations to be created which are more complicated than above examples. It is worth noting, however, that although STREAMS provides the tools, most of the work must be done by the developer of the modules and drivers.

CHAPTER VI.

Implementation of Turing Plus STREAMS

As stated previously, a major part of the project involved the implementation of a Unix STREAMS-like facility in a high-level concurrent language--Turing Plus. A high-level language ensures portability of the code and will allow the facility to be easily integrated into the HM-Nucleus. This chapter describes the environment in which the implementation was carried out and the implementation itself, backing up some of the design decisions that were made along the way.

1. The Environment.

The hardware of the Homogeneous Multiprocessor is still not fully developed. Furthermore, the HM-Nucleus is still likely to undergo many changes before it can be used to support application programs. For these reasons it was decided that any implementation would have to be made in a high-level language to facilitate the later task of integration into the HM-Nucleus. The language chosen was Turing Plus [Holt86], a concurrent version of the Turing language [Holt87], recently developed at the University of Toronto.

Turing is a strongly typed language which is similar to Concurrent Euclid [Holt83]--the language of implementation for the HM-Nucleus. Because of this similarity, translation

of the HM-Nucleus code into Turing Plus would be a trivial task.

Turing offers some features not present in Euclid, which make it the preferred choice of the two. The most prominent such feature is the addition of subprogram (function or procedure) types and variables. This permits procedure and function handles (pointers to subprograms) to be defined, permitting greater flexibility in implementation. As will become evident later, STREAMS relies heavily on this feature in order to achieve a high degree of modularity and flexibility.

Another useful feature of Turing is that its strong type checking can be defeated by using type-casting and by mapping variables of one type onto the address of another. Strict checking usually results in code which contains fewer mistakes, simplifying the debugging stage. However, when the code has been properly debugged, Turing allows the runtime checks to be turned off, making the implementation extremely fast.

The Turing Plus (T+) compiler was installed on a network of Sun workstations running Unix. The workstations are based on a Motorola MC68020 processor, further ensuring the ease of porting to the Homogeneous Multiprocessor. The network can also be used to test protocols by sending messages between Streams on different machines.

2. Overview of Turing Plus STREAMS.

The structure of the code for T+ STREAMS corresponds closely to the hierarchical structure of the Tunis kernel, on which the HM-Nucleus is based. At the topmost level is the **hmnucleus** module, which in the current implementation consists of little more than the declarations of some global variables and constants. For testing purposes a user can be simulated as a process defined at this level. Below **hmnucleus** is the **io** module, which contains some global variables pertaining to the I/O subsystem but not required in the rest of the nucleus. Underneath **io** lies the **streams** module, which holds the actual STREAMS implementation. In reality, STREAMS is itself a two-level hierarchy, with several modules below the **streams** module. These include, among others, modules containing code for STREAMS drivers and other modules, the utilities and STREAMS-specific system calls.

The Turing Plus language incorporates the concept of modules, and also of hierarchies. A module can be defined to be a child of another, by including a **parent** clause. The parent module, on the other hand, will refer to the child in a **child** clause. Thus the **io** module is a child of **hmnucleus** and the parent of the **streams** module.

The relationship between these modules is also reflected in the directory structure of the code files. The root directory is called **hmnucleus**. Below it is **io** and below that the **streams** subdirectories. Below **streams** there are

five subdirectories. Beside the **drivers**, **modules**, and **utils** directories, alluded to above, there are also the **procs** and **allocators** subdirectories. The first one contains the procedures implementing the STREAMS-specific system calls (**getmsg** and **putmsg**, among others). The second one holds the code for procedures which allocate the STREAMS data structures needed to create a Stream. The directory tree, along with a partial listing of files is shown in Figure 7.

3. Data Structures.

T+ STREAMS uses the same main data structures as the original Unix version. The only difficulty encountered in translation was caused by the fact that Turing collections are typed, so that a pointer to a variable in one collection cannot be used to point to a variable in a different collection. The problem with this is particularly evident in the case of the private data structure (pds) pointer in the **queue_t** record. The type of the pds is developer-dependent and not the same for all modules and drivers. The **queue_t** structure, however, is common to all such entities. The chosen solution was to change the field to an address-type, and to store the address of the private data structure, rather than a pointer to it (in Turing a pointer is not just an address). The structure can later be accessed by mapping its record type onto the stored address (written in Turing as: **type_name@(address)**).

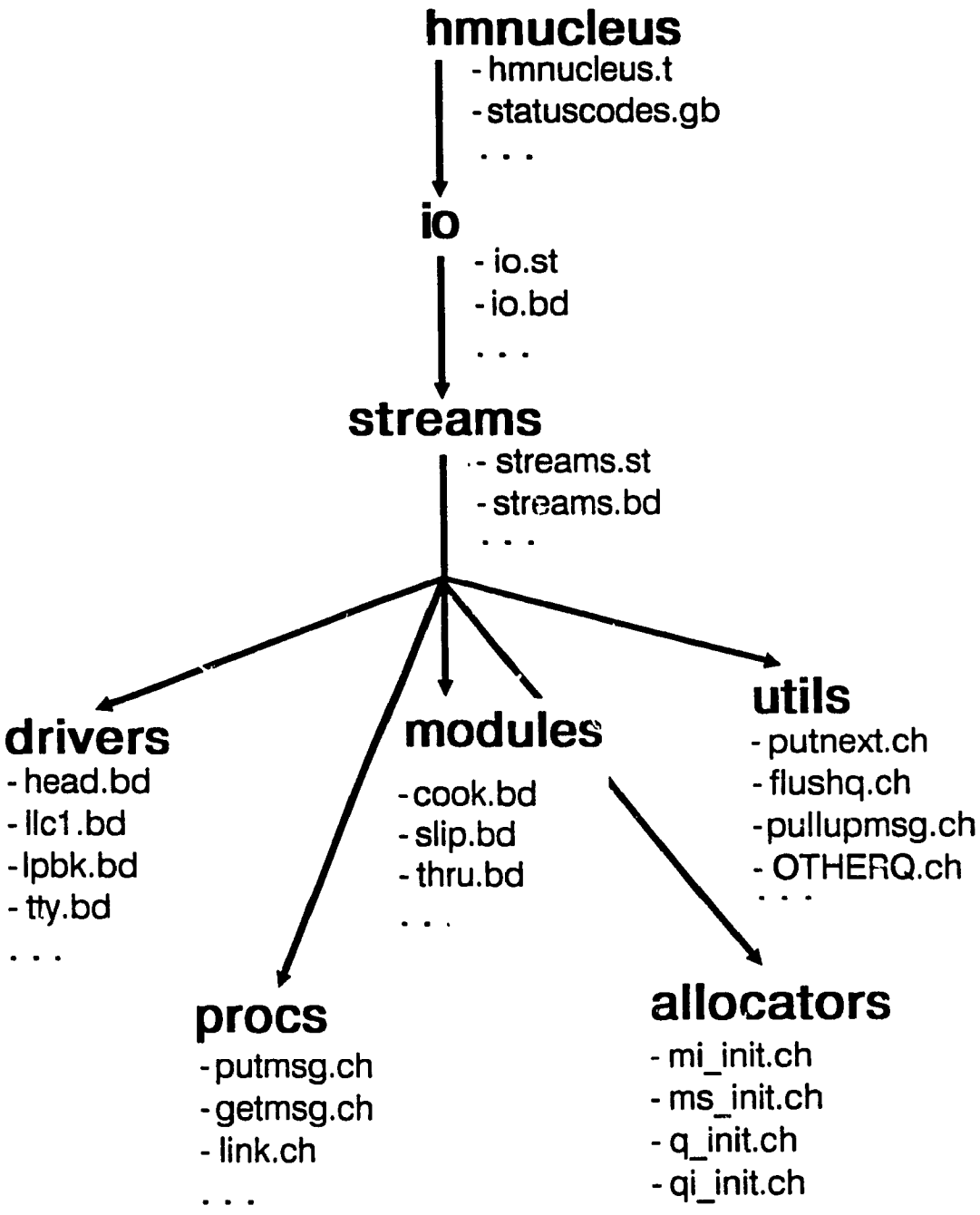

Figure 7. The Turing Plus STREAMS Directory Structure

This solution is adequate for a testing version, but it presents a problem with deallocation. Since the pointer to the pds is not saved, the structure cannot be freed to its collection (this problem does not occur if the pds is allocated from an array). The solution is to use only arrays or unchecked collections in allocating private data structures. In Turing, a pointer to an unchecked collection is equivalent to an address, so it can be stored in the appropriate field of the **queue_t** record.

The **queue_t** structure also had to be altered slightly. The structures are always allocated in pairs, one for each direction on the Stream. A utility routine is provided whereby the other partner of a queue_t structure can be located. Two other routines can locate the read or write side Queue, given either one. These routines rely on the **queue_t** records being allocated at particular memory boundaries. From the address it is possible to determine whether the Queue is an upstream (read) or a downstream (write) Queue, and also to locate the other Queue in the pair. This approach is heavily dependent on the allocation routines of the operating system and is not very portable. Furthermore, Turing does not allow variables to be allocated at desired memory boundaries. Instead, the **queue_t** record has been extended with the **q_other** pointer field which locates the Queue's partner. As far as determining the Queue's directions is concerned, a bit is reserved in the **q_state** field of **queue_t** to denote whether the Queue is on the read or

write side of a module. This bit and the **q_other** field are initialized when the pair of **queue_t** structures are first allocated.

4. Buffer Allocation.

Buffer allocation in STREAMS is handled by the **mem** module, which for reasons mentioned later resides in the **utils** subdirectory. It maintains a pool of buffers whose sizes are varying powers of two. The number of buffers of each size, as well as the range of their sizes are compile-time constants and can be altered according to system requirements. The module also implements the utility procedures used within STREAMS to allocate and release buffer space (**allocb**, **freeb**, **freemsg** and **testb**).

Buffers are maintained in freelists--one for each buffer size. Since the pointer to the next buffer in the list is kept at the beginning of the buffer, sizes less than the machine address cannot be supported. The head pointers for each freelist are kept in an array, indexed by the power of two which determines the size of the buffer.

During the initialization of the module, the overall space needed for the buffer pool is allocated (the number of bytes is the sum of the products of the number of buffers of each size and their respective sizes). Once the logical division of the pool into buffers is performed at initialization, the total number of buffers of any one size remains

constant. No attempt at compaction is made by the buffer manager.

The decision to grant an allocation request depends on the availability of buffer space and the priority of the request. High-priority requests are always satisfied, provided that a buffer of the requested size exists. For medium and low-priority requests, there exist compile-time threshold values, expressed as percentages. The individual thresholds for each buffer size (expressed as number of buffers) are calculated and stored in an array, for quick referencing. A medium-priority request is satisfied provided that the number of buffers of the requested size currently in use has not exceeded the medium threshold value for that size. The same holds for low-priority requests, except that the low threshold value is examined.

At present, the allocating algorithm makes no attempt to satisfy a request with a larger-sized buffer, when no buffers of a requested size are available. A modification to this rule can easily be included, allowing, for example, high-priority requests to be satisfied even when the preferred buffer size is unavailable. The user of the **allocb** utility, of course, is free to request a larger buffer, when the original request is denied.

5. Scheduling.

The scheduling facility in T+ STREAMS consists of two main parts--a scheduler monitor and a scheduling process.

The process, **sched_proc**, resides in the **sched** module, in the **utils** subdirectory. The monitor, **sched_mon**, resides in a module of the same name, also in the **utils** subdirectory. The monitor is declared to be a child of the **sched** module. The code for the buffer manager and the scheduler is included in Appendices F and G, respectively.

The monitor provides exclusive access to the STREAMS scheduler queue, which is manipulated by two exported procedures--**qenable** and **dequeue**. The first is the STREAMS utility routine used to schedule a STREAMS module service procedure for execution. It essentially enters the STRFAMS Queue to be scheduled onto the scheduling queue. The second procedure is used by **sched_proc** to obtain the next Queue to be scheduled.

The function of **sched_proc** is to run the service procedures in the Queues which have been placed on the scheduler queue. With the above setup and the STREAMS data structures, this becomes a trivial task. The process consists of an endless loop which obtains the next Queue to be scheduled using **dequeue**, then invokes the service procedure, which is accessible through the **queue_t** structure which describes the Queue. When the scheduler queue becomes empty, **dequeue** perform a wait on a condition variable. This condition will eventually be signalled by **qenable** when another Queue is scheduled. The waiting must be performed because **sched_proc** cannot exit--it must remain active throughout the lifetime of the system. A condition variable is used because it is

more efficient than allowing the process to busy-wait for the next Queue.

In retrospect, it would be logically more appropriate to place the **mem** module and the scheduler modules into separate directories. Perhaps, the **mem** module belongs in the **allocators** subdirectory because it allocates buffer space. The reason for their present placing is that these modules implement procedures which are STREAMS utilities, and therefore should reside in the **utils** directory.

6. Utilities.

All of the utility routines are contained in the **utils** subdirectory, below **streams**. All of the utilities are children of the **streams** module. Because some utilities make use of others, the order of their declarations in **streams** is important.

Essentially, the utilities are the same as the ones listed in Appendix E, for Unix STREAMS. The major difference is that the Turing Plus compiler does not support macros, so every utility is either a procedure or a function. Also, functions in Turing are not allowed to have side-effects, so the appropriate utilities have been coded as procedures with an extra parameter, corresponding to the returned value.

In addition to the standard utilities two more functions are added. They are **nextsq** and **prevsq**. They are used to locate the next and previous serviceable queue, respec-

tively. A serviceable queue is defined as one which corresponds to a module with a service procedure, and has not been disabled. The **nextsq** routine is used when checking if flow-control is restricting the passing of a message. When the flow-control condition is cleared up, **prevsq** is used to locate a queue to be back-enabled.

Utilities which deal with buffer management (**allocb**, **freeb**, **freemsg**, and **testb**) are placed together in a separate module--**mem**. The procedures are exported from **mem** unqualified, which means that the same naming convention can be used for these and all other utility routines.

The **qenable** procedure, which deals with scheduling, is similarly contained within the **scheduler** module. The code for the buffer manager and the scheduler is included in appendices F and G, respectively.

Two utility routines from Unix STREAMS--**splstr** and **strlog**--are not included in this implementation. The first is used to raise to the interrupt level when a module is executing a critical section of code. Turing Plus supports monitors which can be used to implement critical sections, so the routine is not necessary. The second routine is used to submit a message to the logging driver. Because message-logging is not an essential part of STREAMS it is not supported in this implementation, rendering the **strlog** routine unnecessary.

Another routine--**buffcall**--is included in the **utils** directory but does not contain any code. In the original

implementation it provides a method of recovering from buffer allocation failures. When **allocb** fails, the **buffcall** utility is invoked. The parameters to **buffcall** include the priority and size of the failed request. They also provide a function, and an argument to the function, which will be invoked when a buffer of appropriate size at the given priority becomes available.

Two examples of the use of this utility were given. In one case, the function invoking **buffcall** was the driver receive interrupt handler. When **allocb** failed **buffcall** was invoked, specifying the handler as the function to be called later. In the second example, **buffcall** was invoked by a service procedure, again specifying itself as the function to be called. C does not differentiate between procedures and functions, nor does it complain about incompatible function types. The function in the parameter list of **buffcall** is specified as returning an integer and having one integer parameter. The actual function passed to **buffcall**, on the other hand, may be different in various cases. While C allows such flexibility, the strong type-checking of Turing makes this implementation impossible. Clearly, this matter requires some further exploration.

For the time being, at least, buffer exhaustion may be handled as follows. In the first case, when driver input data cannot be copied into a message, the data are simply discarded. This would still happen even if **buffcall** were implemented, when more input arrived before buffer space

became available. In the case of a **service** procedure, the message should be discarded as well. The **service** procedure cannot simply put the message back on the queue as in the flow control case. There is no guarantee that another message will arrive, so the Queue may never be scheduled. Another possibility is for the **service** procedure to put the message back and continue processing the other messages. Some of these messages may be consumed locally and their buffer space released, allowing the previously returned message to be processed. The **putbq** utility could not be used for this purpose, as it places messages at the head of the queue. A new utility would be needed, which would place these messages at the end of the queue to delay their processing. It would also be necessary to keep track of which messages have been put back for delay purposes, as the case can happen when all messages in the queue are being delayed, and the service procedure would repeat the process of getting a message, putting it at the back of the queue, and getting the next message. Beside these complications, this plan almost certainly would cause messages to arrive out-of-sequence, which may pose problems.

7. A Sample Protocol Module.

Several T+ STREAMS modules and drivers have been coded and reside in appropriate subdirectories. Most are translations of examples given in the **STREAMS Programmer's Guide** [AT&T87b]. Translating these modules provided invalu-

able help in understanding the fundamental aspects of STREAMS as well as giving an insight into particular features such as scheduling and multiplexing. In this section we present the structure of a T+ STREAMS pseudo-driver, which implements the services of the IEEE 802.2 standard for Class I Logical Link Control (LLC-I) [ANSI87, IEEE85]. The standard provides a connectionless, unacknowledged service.

The LLC module is structured after the sample multiplexor in the **STREAMS Programmer's Guide** [AT&T87b], but it is by no means a translation. The code for the module, given in Appendix H, consists of three files in the **drivers** subdirectory. One file contains the declarations of types and constants needed for the module. This file is included in the **streams** module, making the declarations visible throughout STREAMS. The two other files are the stub and body of the LLC module. Stub files are used in Turing Plus to allow separate compilation. The LLC-I driver, as all other STREAMS drivers and modules, is a child of the **streams** module.

The upper part of the LLC-I multiplexor consists of only the upper write-put procedure. The Lower part consists of the lower write-service procedure, and both the put and service procedures of the lower read side.

The upper write-service and the lower write-put procedures are not used because the upper-put and lower-service procedures of the write side complement each other. That is

to say, the put procedure enqueues the message, which is subsequently processed by the service procedure.

The upper read-side procedures are both absent because the lower side puts upstream messages directly into one of the upper Streams. The upper read Queue structure is used only as a reference point for the **putnext** utility.

The module's open procedure supports multiple minor device numbers and the CLONEOPEN option. This option is used when no specific minor device number is given by the user, and the module is requested to automatically select some available number. The first time **open** is called the minor device number must be zero (if CLONEOPEN is specified it is set to zero). The zero minor device number is intended to be associated with the controlling Stream, used by the driver daemon or other controlling process.

The linking information structure is allocated dynamically in the **open** procedure, and not from an array. Because of this the number of upper Streams is not directly limited. However, each upper Stream is associated with an SAP address. The SAP address size is one byte, and two bits are reserved. This allows 32 SAP addresses, and hence 32 upper Streams, including the controlling Stream. In reality, some of these addresses are already reserved by IEEE for particular protocols, but in this implementation the entire address space is used when selecting an address for the CLONEOPEN option. The function which performs this selection is appropriately named **next_sap**.

Every time a new upper Stream is opened, a special SAP linking structure is allocated. This structure holds a pointer to the upper Queue and the SAP address of the upper Stream (this is the same as the minor device number). These structures are linked together to form a list of SAP's, which is used by the lower part to locate downstream, and route upstream messages.

The upper Queues are serviced in a round-robin fashion by the lower write-service procedure. The **get_next_q** procedure returns the next upper Queue that needs servicing, or a nil pointer, if all Queues are empty. A global variable keeps track of the last Queue serviced, so that round-robin scheduling can proceed fairly in those cases when the service procedure is unable to service all Queues during one invocation.

Upstream messages are placed by the lower procedures directly into the Queue immediately above the appropriate upper Queue, specifying that upper Queue as a parameter to **putnext**. The upper Queue is located by matching the desination SAP address in the message with the address field of one of the records in the SAP list. This search is performed by the **find_sap** routine.

If the destination address is a group address, the function **find_group** is called instead of **find_sap**. Although group addressing is not fully supported the intent is that this routine will return a pointer to another list of SAP records, each of which belong to the same group. A nil

pointer is returned if the group has no membership at a particular station. The manner in which groups are created and maintained is a matter for further investigation. At present, the **find_group** function always returns a nil pointer.

The 802.2 standard logically divides the LLC into a Station Component part and one or more SAP Component parts. The station component handles all primitives which affect the station as a whole. The SAP component handles all primitives which affect only a particular SAP. Each SAP component is addressed by its corresponding SAP address, while the station component is addressed by the NULL SAP address. In the STREAMS implementation, this logical division maps very closely onto the division of the multiplexor into a lower and upper parts. The upper part, handling each of the upper Streams, represents the SAP components, while the lower part performs the actions of the station component. The only deviation from this is the fact that M_IOCTL messages, used for linking and unlinking lower Streams, are processed by the upper write-put procedure. This is done because before a lower Stream is linked, there is no lower Queue, hence no way of scheduling the lower write-service procedure. However, only the controlling Stream is allowed to perform linking actions, and the controlling Stream--as is the Station Component--is addressed as SAP zero.

Since STREAMS is message-based rather than procedure-based, all of the 802.2 primitives (requests, responses,

indications and confirmations) are implemented as messages. Two message types--M_PROTO and M_PCPROTO--are used to represent these primitives. In addition to these, the LLC performs standard driver and head flush-message processing (because it is a pseudo-driver, the upper part acts like a Stream end of the upper Streams, and the lower part acts as the Stream head of the lower Stream). Also, since it is a multiplexor, it processes M_IOCTL messages (as mentioned above).

The protocol-specific messages (M_PROTO and M_PCPROTO) are structured as follows. The first message buffer contains an LLC header record (defined in **llc1_types.in**). This corresponds to the actual header information, consisting of the source and destination station addresses, the source and destination SAP addresses, the control field and the priority and service class fields. The last two fields are not used by the current implementation, because provision of different priorities and service-classes depends on the medium-driver. A header is considered valid if it is large enough to contain the first five fields (declared as **min_header_size** in **llc1_types.in**). The second (and any additional) message block contains the data part of TEST or UI (Unnumbered Information) messages (the 802.2 primitives--such as UI, TEST, XID and others--are defined in [ANSI87] and [IEEE85]).

M_PROTO messages are used to send protocol messages (UI, XID and TEST). M_PCPROTO messages are used to pass

certain request and confirmation primitives, as outlined below.

The Network-layer, or any user of the LLC module, must adhere to the following interface. Downstream M_PROTO messages are only used to send data (UI in 802.2 terminology). All four address fields of the header, as well as the priority (if used) must be provided. The control field will be filled in by the LLC module. Downstream M_PCPROTO messages are used to request the sending of TEST or XID messages, to request the activation or deactivation of the SAP component, or to put the station component in an up or down state (the last two functions are reserved for the controlling Stream). All fields in the header must be provided. The control field can take on the value of TEST_req, XID_req, ACTIVATE_req or DEACTIVATE_req (these values are defined in **llc1_types.in**). Only the TEST_req message can have any data part. In addition to the above, the controlling Stream is allowed to specify UP_req or DOWN_req in the control field, to change the state of the station component. Such requests from any other Stream will be discarded.

Upstream M_PROTO message, coming from the Physical-layer module, carry UI, TEST or XID messages. UI messages can only be sent as commands, while the TEST and XID messages can be commands, or responses to previously sent TEST or XID command messages. Upstream M_PCPROTO messages are used to represent the transmission status confirmation primitive.

This primitive is not defined in [IEEE85] but appears in an update to the standard [ANSI87]. It is not specified how the primitive should be used, or for what purpose. Class I LLC's do not provide message retransmission, so this primitive should not be used for this purpose. It could be used by the Physical Layer to report a failure of the communication medium, but without any primitive to report this failure to higher layers the Class I LLC simply discards this message type.

The upper Stream is not required to send back TEST and XID responses, but it should be noted that the SAP component part will not provide these replies automatically. In this fashion, TEST and XID messages can be used by higher-level protocols to implement other functions as appropriate.

CHAPTER VII.

Overview and Conclusions.

1. Overview.

The HM possesses a unique architecture, with two separate and rather distinct communication paths. The extended bus mechanism for sharing memory is a novel approach. While most shared-memory systems provide a global memory module accessible through a cross-bar switch, the HM extends its pipeline-like structure by supporting only nearest neighbor memory sharing. This is done because the HM is not intended to be a general-purpose processor, but a powerful computing engine for pipelined algorithms, such as are commonly used in digital signal processing.

In order for the tight coupling present in the hardware to be prominent at the application level, the communication protocols used have to be relatively uncomplicated, since heavy-weight protocols--such as TCP/IP--are generally too slow. At the same time, the duality of the communication paths means that the chosen protocols must be flexible enough to allow different media at the physical level. The IEEE 802 protocol suite was chosen for this reason, and, also, because it is a recognized standard. The 802.2 Link level protocol is designed to interface with several Physical level protocols--a required feature in this project, since both the H-Network and shared memory media must be supported. Because the H-Network design is similar to that

of the Ethernet, the 802.3 protocol can easily be adapted as the physical level protocol for the H-Network. Although no standard exists for shared memory communications, a new protocol can be designed relatively easily. A proposed protocol which will fit well into the 802 protocol suite was presented in Chapter IV.

Li's original proposal for the memory management system is insufficient, as it creates a cyclic dependency. This difficulty was explored and a solution was proposed.

Based on the requirements of the Homogeneous Multiprocessor and its proposed applications, and taking into account the problems with the original memory management system, the proposed structure for the Communications Subsystem was presented. STREAMS was chosen for the implementation of this subsystem due to its flexibility and expandability. STREAMS allows access to virtually any level in the ISO/OSI hierarchy, and can be expanded dynamically to include new media and protocols. Another reason for the choice is that STREAMS is likely to become a standard communication system for Unix.

A proposal for the design of channel devices was also presented. This design satisfies the requirements of both the upper level algorithms and the memory management of STREAMS and the HM-Nucleus.

The operation of the original STREAMS facility was summarized. Next, a detailed description of the STREAMS implementation for the Homogeneous Multiprocessor was pre-

sented. This implementation was written in Turing Plus--a concurrent language with strong type checking, modular structure, code reusability and almost all of the flexibility of C. As an example of STREAMS programming, the code for a module implementing the 802.2 standard for a Class I Logical Link Controller was included. Because STREAMS provides developers with many "building blocks", the module itself required only three days to complete.

2. Future Work.

Although the Turing Plus implementation STREAMS is fully operational, the hardware for the Homogeneous Multiprocessor is not available for testing. Testing the functionality of STREAMS was carried out using a loop-back driver and a special-purpose Stream head. This special Stream head is a module which accepts commands from the operator to send and receive messages of various sizes on the Stream to which it is connected. The data for the downstream messages are taken from a file specified when the Stream is created. Likewise, data from upstream messages received by the Stream head are written to another file. The loopback driver at the downstream end of the Stream is the Turing version of an example given in The STREAMS Programmer's Guide. It receives downstream messages and simply reroutes them back upstream.

This configuration allows the basic functionality of STREAMS to be tested. It is also possible to push interme-

diate modules on the Stream. These modules can implement communications protocols or act as simple filters. A careful selection of input data would allow for more rigorous testing of both the modules as well as the STREAMS utilities.

Another future project would be the implementation of a STREAMS module implementing the shared memory protocol presented in chapter IV. Because the hardware is unavailable, the functionality of shared memory would have to be simulated in software. The simplest approach is to implement a common buffer shared by two (or more) processes. Each process will represent the shared memory driver of a neighbor processor. The **poke** operation can be simulated by a signal on an appropriate condition. One common buffer and three condition variables can be used to represent three-way shared memory communication.

The functionality of the channel device can also be simulated in software in a similar fashion. The common buffer in this case would only have to be big enough to hold the channel information, such as the channel id, in-use flag, and message buffer address. Two such buffers would be needed for every process representing the channel device driver of a neighboring processor--one for communicating with left neighbor and one for the right. No memory transformation would have to be done to the message buffer address, since all processes would be running on the same machine.

When STREAMS modules representing the shared memory and channel device drivers are in place, the full functionality of STREAMS can be displayed. A Stream can be created which would multiplex upstream messages between several Stream heads, and route downstream messages between the shared memory, LLC-I or other driver. Further work would also include development of STREAMS modules to perform the functions of the UDS and RPC layers of the Communications Subsystem of the HM-Nucleus.

3. Conclusions.

As was stated earlier, the Homogeneous Multiprocessor has a unique architecture, where two very different communication paths exist. Furthermore, since the Multiprocessor is intended to be used as a special-purpose computing engine for applications such as digital signal processing, the communications layer of the operating system must provide fast, uncomplicated access to both media--and it is desirable that this access be uniform. We have shown that the above can be accomplished in very standard ways, by adapting a standard protocol and framework. We also note that STREAMS provides a very flexible communications framework. The communication layer can be tailored to the specifications of a particular algorithm without major reconstruction and providing the same interface to users. By allowing modules to be pushed onto a Stream dynamically, STREAMS allows the Communication subsystem to be configured without

the need to restart the system. By allowing any Streams module to be directly below the Stream head, a Communication layer built within the STREAMS framework can easily provide a uniform interface across all levels of the ISO/OSI hierarchy.

Turing Plus was shown to be an excellent systems programming language as it included most of the flexibility of C, while being very structured and strongly type-checked.

BIBLIOGRAPHY

[ANSI87] American National Standards Institute, Revised Text of ISO/DIS 8802/2 for Second DIS Ballot, ANSI Document number ISO/TC 97/SC 6 N4453.

[AT&T87a] AT&T, STREAMS Primer, Prentice Hall, 1987.

[AT&T87b] AT&T, STREAMS Programmers Guide, Prentice Hall, 1987.

[Blyt84] David Blythe, Peter Ewens, Mark Funkenhauser, Mark Hume, **"The Structure of the Tunis Operating System"**, Computer Systems Research Institute, University of Toronto, May 1984.

[Brow82] D.R. Brownbridge, L.F. Marshall, B. Randell, **"The Newcastle Connection or UNIXes of the World Unite!"**, Software: Practice and Experience, Vol. 12, 1982.

[Brow84] Robert L. Brown, Peter J. Denning, Walter F. Tichy, **"Levels of Abstraction in Operating Systems"**, IEEE Computer, October 1984.

[Brow85] Robert L. Brown: personal communication to J.W. Atwood.

[Chan84] Samuel T. Chanson, K. Ravindran, Stella Atkins, **"Performance Evaluation of the ARPANET Transmission Control Protocol in a Local Area Network Environment"**, Technical Report 85-6, UBC, December 1984.

[Chan85] Samuel T. Chanson, K. Ravindran, Stella Atkins, **"LNTP - An Efficient Transport Protocol for Local Area Networks"**, Technical Report 85-4, UBC, February 1985.

[Cher84] David R. Cheriton, **"The V Kernel: A Software Base for Distributed Systems"**, IEEE Software, April 1984.

[Cher88] David R. Cheriton, **"The V Distributed System"**, Communications of the ACM, Vol. 31, #3, March 1988.

[Ches87] Greg Chesson, **"Protocol Engine Design"**, Proceedings of the 1987 USENIX Technical Conference, Phoenix, Arizona, June 1987.

[Dimo83] Nikitas J. Dimopoulos, **"The Homogeneous Multiprocessor Architecture - Structure and Performance Analysis"**, Proceedings of the 1983 International Conference on Parallel Processing, August 1983.

[Dimo85] Nikitas J. Dimopoulos, **On the Structure of the Homogeneous Multiprocessor"**, IEEE Transactions on Computers, vol. C-34, #2, February 1985, pp. 141-150.

[Dimo87] Nikitas J. Dimopoulos, Kin Fun Li, Eric Chi Wah Wong, D.V. Ramanamurthy, J. William Atwood, **"The Homogeneous Multiprocessor System: An Overview"**, Proceedings of the 1987 International Conference on Parallel Processing, The Pennsylvania State University, August 17-21, 1987, pp.592-599.

[Holt83] Richard C. Holt, **Concurrent Euclid, the UNIX System, and Tunis**, Addison-Wesley Publishing Company, 1983.

[Holt86] R. C. Holt and J.R. Cordy, **The Turing Language Report**, Technical Report CSRI-153, Computer Systems Research Institute, University of Toronto, December 1983, last updated August 1986.

[Holt88] Richard C. Holt and J.R. Cordy, **The Turing Plus Report**, Technical Report CSRI-214, Computer Systems Research Institute, University of Toronto, August 1988.

[IEEE85] IEEE, **Logical Link Control 802.2**, 1985.

[Jone79] Anita K. Jones, Robert J. Chansler, Jr., Ivot Durham, Karsten Schwans, Steven R. Vegdahl, **"StarOS, a Multiprocessor Operating System for the Support of Task Forces"**, Proceedings of the 7th Symposium on Operating Systems Principles, SIGOPS, Vol. 10, #12.

[Li87a] Kin Fun Li, Nikitas J. Dimopoulos, J. William Atwood, **"The HM-Nucleus: A Distributed Kernel Design for the Homogeneous Multiprocessor"**, IEEE Micro, February 1987.

[Li87b] Kin Fun Li, **The Homogeneous Multiprocessor: a Simulation Study and an Operating System Design**, PhD Dissertation, Concordia University, Department of Electrical Engineering, July 1987.

[Mend83] Mark P. Mendell, **"Structure of a Portable Operating System"**, Master's Thesis, University of Toronto, Department of Computer Science, 1983.

[Metc76] Robert M. Metcalfe, David R. Boggs, **"Ethernet: Distributed Packet Switching for Local Computer Networks"**, Communications of the ACM, Vol. 19 #7, July 1976.

[Oust80] John K. Ousterhaut, Donald A. Scelza, Pradeep S. Sindhu, **"Medusa: An Experiment in Distributed Operating System Structure"**, Communications of the ACM, Vol. 23 #2, February 1980.

[Oust87] John Ousterhaut, Andrew Cherenson, Fred Douglis, Michael Nelson, Brent Welch, **"An Overview of the Sprite Project"**, ;login:, Vol. 12, #1, Jan/Feb 1987.

[Panz85] Fabio Panzieri, Design and Development of Communication Protocols for Local Area Networks, PhD. Dissertation, University of Newcastle upon Tyne, 1985.

[Pisc86] David M. Piscitello, Alan J. Weissberger, Scott A. Stein and A. Lyman Chapin, "Internetworking in an OSI environment", Data Communications, May 1986, pp. 118-136.

[Pope81] G. Popek, B. Walker, J. Chow, D. Edwards, C. Kline, G. Rudisin, G. Thiel, "LOCUS - A Network Transparent, High Reliability Distributed System", Proceedings of the 8th Symposium on Operating System Principles, December 1981.

[Sega88] Terry Segal, The Homogeneous Multiprocessor Memory Modules and the Interbus Switch Controller, M.Eng. Thesis, Concordia University, Department of Electrical and Computer Engineering, 1988.

[Solo79] Marvin H. Solomon, Raphael A. Finkel, "The Roscoe Distributed Operating System", ACM 1979.

[Tane81] Andrew S. Tanenbaum, Computer Networks, Prentice Hall Inc., 1981.

[Wong85] Eric Chi Wah Wong, A Collision Free Protocol for LANs Utilizing Concurrency for Channel Contention and Transmission", M. Eng. Thesis, Concordia University, 1985.

[Weis87] Alan J. Weissberger, Jay E. Israel, "What the New Internetworking Standards Provide", Data Communications, February 1987.

[Welc86] Brent B. Welch, "The Sprite Remote Procedure Call System", July 1986.

[Wulf74] W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, F. Pollack, "HYDRA: The Kernel of a Multiprocessor Operating System", Communications of the ACM, Vol. 17, #6, June 1974.

Appendix A.

The Switch Closing Algorithm.

A switch can exist in one of three logical states, and a transition to the next state is governed by the operational Algorithm 1.2 [Dimo83]. The closing of a switch is carried out in two stages. During the first stage, the next logical state is determined, while the physical closing of the switch is performed during the second stage. The three logical states in which a switch can exist are as follows:

OPEN: This state signifies that no request to close a switch exists, or that if a request exists, it will not be honored immediately because a neighboring switch is currently servicing a request.

GRAY: This state signifies that a request is acknowledged and switch closure will be carried out in the immediate future.

CLOSED: This state signifies that it is safe for a switch to close, and stage two can proceed immediately.

The Operational Algorithm 1.2, which governs the state transition of a switch is as follows:

Algorithm 1.2.

For a switch S_i:

If no request exists, it becomes **OPEN**,

Otherwise, if a request exists, then:

> If **OPEN** it becomes **GRAY**, provided that the switch to its left, S_{i-1}, is **OPEN**, otherwise it remains **OPEN**.
>
> If **GRAY**, it becomes **CLOSED**, provided that the switch to its right, S_{i+1}, is **OPEN**, otherwise it remains **GRAY**.
>
> If **CLOSED**, it remains **CLOSED**.

The left-most S_0, and right-most, S_k, switches are always **OPEN**.

Appendix B.

The ISO/OSI Network Hierarchy.

Level 1. The Physical Layer

This layer is concerned with transmitting raw bits over a communication channel. The principal problem is making sure that when a one is sent at one end, the other end also receives a one, not a zero. Typical design decisions involve how many volts should represent a one or a zero, and how long each bit should be (transmission rate).

Level 2. The Data Link Layer.

The function of the Data Link Layer is to take a raw transmission facility and transform it into a line which appears to be free of transmission errors. Questions addressed at this level include frame format, acknowledgements, flow-control, retransmission and duplicate detection. All of the above are at the link, or point-to-point level only. Acknowledgements and flow control at the end-to-end level are handled by a higher layer.

Level 3. The Network Layer.

The Network Layer is concerned with the operation of the subnet. Particular questions addressed here include routing decisions and congestion control. Inter-network flow control (between Gateways) is also handled by the

Network Layer. This layer is only present in internetworking systems.

Level 4. The Transport Layer.

This layer is concerned with the end-to-end aspects of the communication. End-to-end flow control, acknowledgements, as well as retransmission and detection of duplicates are some of the problems dealt with here. The set-up, maintenance and break-down of connections also concern this level. Fragmentation may also have to be performed.

Level 5. The Session Layer.

The Session Layer is generally considered to be the user interface to the network. It is here that functions such as user authentication and permission checking are performed. Another duty of this layer is to recover gracefully from breakdowns of the underlying network.

Level 6. The Presentation Layer.

This level is mainly a data-transformation layer. Functions include text compression, conversion between data representation on different machines, and data encryption and decryption.

Level 7. The Application Layer.

The structure of the Application Layer is up to the individual user. A protocol at this level defines what kind

of communication goes on between individual users. Some questions addressed at this level, however, are general. These include network transparency and problem-partitioning. A typical Application Layer protocol is the Remote Procedure Call.

Appendix C.

STREAMS Data Structures.

Structures defining a module or driver.

```
struct streamtab {
  struct qinit *st_rdinit;   /* defines read QUEUE */
  struct qinit *st_wrinit;   /* defines write QUEUE */
  struct qinit *st_muxrinit; /* for multiplexing drivers only */
  struct qinit *st_muxwinit; /* for multiplexing drivers only */
};

struct qinit {
  int    (*qi_putp)();    /* put procedure */
  int    (*qi_srvp)();    /* service procedure */
  int    (*qi_qopen)();   /* called on each open or a push */
  int    (*qi_qclose)();  /* called on last close or pop */
  int    (*qi_qadmin)();  /* reserved for future use */
  struct module_info *qi_minfo; /*information structure */
  struct module_stat *qi_mstat; /* statistics structure -- optional */
};

struct module_info {
  ushort  mi_idnum;   /* module ID number */
  char   *mi_idname;  /* module name */
  short   mi_minpsz;  /* min packet size accepted, for developer use */
  short   mi_maxpsz;  /* max packet size accepted, for developer use */
  ushort  mi_hiwat;   /* hi-water mark, for flow control */
  ushort  mi_lowat;   /* lo-water mark, for flow control */
};

struct module_stat {
  long   ms_pcnt;   /* count of all calls to put procedure */
  long   ms_scnt;   /* count of all calls to service procedure */
  long   ms_ocnt;   /* count of all calls to open procedure */
  long   ms_ccnt;   /* count of all calls to close procedure */
  long   ms_acnt;   /* count of all calls to admin procedure */
  char  *ms_xptr;   /* pointer to private statistics */
  short  ms_xsize;  /* length of private statistics buffer */
};
```

QUEUE Structure

```
struct queue {
  struct qinit *q_qinfo;  /* procedures and limits for queue */
  struct msgb  *q_first;  /* head of message queue for this QUEUE */
  struct msgb  *q_last;   /* tail of message queue for this QUEUE */
  struct queue *q_next;   /* next QUEUE in Stream */
  struct queue *q_link;   /* next QUEUE on STREAMS scheduling queue */
  caddr_t       q_ptr;    /* pointer to private data structure */
  ushort        q_count;  /* weighed count of chars on message queue */
  ushort        q_flag;   /* QUEUE state */
  short         q_minpsz; /* min packet size accepted by this QUEUE */
  short         q_maxpsz; /* max packet size accepted by this QUEUE */
  ushort        q_hiwat;  /* message queue high water mark */
  ushort        q_lowat;  /* message queue low water mark */
};
typedef struct queue queue_t;
```

Structure of STREAMS Messages.

```
struct     msgb {
  struct   msgb  *b_next;   /* next message on queue */
  struct   msgb  *b_prev;   /* previous message on queue */
  struct   msgb  *b_cont;   /* next message block of message */
  unsigned char  *b_rptr;   /* first unread data byte in buffer */
  unsigned char  *b_wptr;   /* first unwritten data byte in buffer */
  struct   datab *b_datap;  /* data block */
};
typedef struct msgb mblk_t;

struct     datab {
  struct   datab *db_freep; /* used internally */
  unsigned char  *db_base;  /* first byte of buffer */
  unsigned char  *db_lim;   /* last byte+1 of buffer */
  unsigned char   db_ref;   /* count of messages pointing to block */
  unsigned char   db_type;  /* message type */
  unsigned char   db_class; /* used internally */
};
typedef struct datab dblk_t;
```

Appendix D.

STREAMS Message types.

STREAMS defines eighteen message types--nine each of ordinary and high priorities. A STREAMS module or driver can generate any type of message or change the type of any message, and send these messages in any direction on the Stream. However, certain rules governing the use of some message types, although not enforced, should be observed.

The following list briefly describes each message type and its intended application. The list is divided into two parts--one for ordinary and one for priority message types. Priority messages are always enqueued ahead of ordinary messages, but otherwise their processing has no special characteristics.

Ordinary Message Types.

M_DATA: Contain ordinary data. This is the default message type when buffers are allocated by **allocb**. The contents of the M_DATA message blocks are passed as the data part to the user invoking the **getmsg** system call; a call to **putmsg** copies the user's data part into an M_DATA block.

M_PROTO: Intended to contain control information, such as protocol header and/or trailer fields. The control part in a call to **getmsg** or **putmsg** will be associated with this

message type. A typical protocol interface message will contain one M_PROTO (or M_PCPROTO, described later) block, containing the interface parameters, and one or more M_DATA blocks, containing the protocol service data unit (psdu).

M_IOCTL: Generated by the Stream head when an **ioctl** system call is issued on the Stream. The M_IOCTL block will contain an **iocblk** structure, which contains information identifying the specific **ioctl** command and the user, and specifying whether any data follows. This message is acted upon by the first module (or driver) which understands it. The Stream head expects an acknowledgement to be returned with an **M_IOCACK** or **M_IOCNAK** message.

M_CTL: Typically used for inter-module communication, as, for example, when adjacent protocol modules negotiate the terms of their interface. Cannot be generated by the user and discarded if received at the Stream head.

M_BREAK: A special case of the M_CTL message, used to request a driver to send a BREAK signal on whatever medium it accesses. Cannot be generated by user and always discarded when passed to a Stream head.

M_DELAY: Sent to a driver to request output delay. The buffer of this message type is expected to contain an integer indicating the length of the delay in machine ticks.

Typically intended to prevent swamping slower devices. This message cannot be generated by a user and is discarded if passed to the Stream head.

M_PASSFP: Used to pass a file pointer from one end of a pipe Stream to the other end. A pipe Stream is defined as one which is terminated at both ends by a Stream head. This message is generated as a result of a special ioctl call on the sending Stream head, and is placed directly into the receiving Stream head, without passing through the Stream.

M_SETOPTS: Used by a downstream module to alter some characteristics of the Stream head. The data buffer of this message contains a special structure describing the options to be set and their new values. Options include initial write offset for data buffers, min and max packet sizes, and high and low water marks. Interpreted only by the Stream head and passed unchanged by other modules.

M_SIG: Sent by downstream module or driver to post a signal to a process. The type of signal is stored in the first byte of the data buffer. If the Stream is a controlling TTY for its process group and the signal is not SIGPOLL, the signal is sent to the entire group. Otherwise, the signal is sent to those processes which have registered to receive that signal.

Priority Message Types.

M_PCPROTO: Same as the M_PROTO type, except that priority enqueuing is performed. Also, when this message type is placed on a message queue, the corresponding Queue is always enabled. Only one M_PCPROTO message is allowed to be in a message queue at any one time--all others will be discarded by the Stream head. Intended to allow data and control information to be sent unaffected by flow control.

M_ERROR: Sent upstream by a module or driver to report some error condition. When a Stream head receives this message it becomes locked. This means that all calls to the Stream, other than **close** and **poll**, will fail, with the global variable **errno** set to the first data byte. All processes sleeping on calls to the Stream are awakened and the Stream is flushed.

M_HANGUP: Sent by a driver to report that it can no longer send data upstream. When the message reaches the Stream head the Stream is marked so that all calls resulting in a downstream message fail, returning ENXIO.

M_IOACK: This is the positive acknowledgement of a previous M_IOCTL message. May contain information from the sending module or driver, which the Stream head returns to

the user if a corresponding outstanding M_IOCTL request exists.

M_IOCNAK: Same as above, only the acknowledgement is negative. This message signals the failure of the corresponding **ioctl** call.

M_FLUSH: Request to all modules and drivers to discard messages on the corresponding message queues. The first byte specifies whether the read, write or both sides of the Stream are to be flushed. Each module flushes its queue, then passes the message on. The Stream head or driver may be required to route the message in the opposite direction, depending on the flag.

M_PCSIG: Priority version of M_SIG.

M_START and M_STOP: Used to request a driver to start or stop their output. Not intended to turn devices on or off, but rather to produce pauses in output. These messages cannot be generated by a user and are discarded if passed to the Stream head.

Appendix E.

STREAMS Utilities.

The following is an alphabetized list of the utilities available within the Unix STREAMS facility. The declaration of each utility and its parameters is first given, followed by a short description of its function. For a more detailed description refer to Appendix C of the STREAMS Programmer's Guide [AT&T87b].

```
int adjmsg(mp, len)
mblk_t *mp;
int len;
```

Trim +len bytes from the beginning, or -len bytes from the end, of message pointed at by **mp**. Only trims bytes across message blocks of the same type, fails if there are not enough bytes of similar type.

```
mblk_t *allocb(size, pri)
int size, pri;
```

Returns a pointer to a message block of type M_DATA with a buffer of at least **size** bytes, if such a buffer at the indicated priority, **pri**, exist. A nil pointer is returned if buffer space is unavailable.

```
queue_t *backq(q)
queue_t *q;
```

Returns a pointer to the Queue behind the given Queue, or a nil pointer if no such Queue exists (as when **q** is the Stream end).

```
int buffcall(size, pri, func, arg)
int (*func)();
int size, pri;
long arg;
```

Assists in recovering from buffer allocation failure. When **allocb** returns a nil pointer, the calling function can use **buffcall** to reschedule itself when the proper-sized buffer at the indicated priority becomes available. If **buffcall** returns a one, then **func()** will be called with the argument **arg**; a return value of zero indicates a temporary inability to allocate internal structures, and **func** will not be called.

```
int canput(q)
queue_t *q;
```

Determine whether the message queue of **q** is full or not (as determined by the high-water mark). If **q** has no service procedure associated with it, the Stream is searched until such a Queue is found or an end is reached. A one is returned if **q** was not full (or if the search stopped at one of the Stream's ends); a zero is returned otherwise, causing the caller to be **blocked.**

```
mblk_t *copyb(bp)
mblk_t *bp;
```

Makes a copy of the message block pointed at by **bp** and returns a pointer to the new block. The new block is allocated with medium priority. If the buffer cannot be allocated (i.e., **allocb** returns a nil pointer), then a nil pointer is returned.

mblk_t *copymsg(mp)
mblk_t *mp

Uses **copyb** to make a copy of a whole message chain, pointed at by **mp**. Returns a pointer to the new message only if all buffers could be allocated, otherwise all allocated blocks are freed and a nil pointer is returned.

#define datamsg(mp) ...

This macro returns TRUE if **mp** (declared as **mblk_t ***) points to a message whose first block is of type M_DATA, M_PROTO or M_PCPROTO; otherwise it returns FALSE.

mblk_t *dupb(bp)
mblk_t *bp;

The message block pointed at by **bp** is duplicated, that is a new **mblk_t** is allocated and linked to the original data block (**dblk_t** structure). The reference count in the data block is incremented. A pointer to the new message descriptor is returned on success. If the message descriptor could not be allocated, a nil pointer is returned instead.

mblk_t *dupmsg(mp)
mblk_t *mp;

A whole message is duplicated using successive calls to **dupb**. Returns a pointer to the new message or a nil pointer if any call to **dupb** failed.

#define enableok(q) ...

This macro cancels the effect of an earlier **noenable** call. It allows **q** (declared as **queue_t ***) to be serviced by the STREAMS scheduler.

```
int flushq(q, flag)
queue_t *q;
int flag;
```

Removes and frees messages from the message queue of **q**. The flag determines whether all messages, or only data messages are to be flushed. If a Queue behind **q** had been blocked, **flushq** may perform back-enabling (as described in the section on flow control).

```
int freeb(bp)
mblk_t *bp;
```

Frees the message block pointed at by **bp**. The message block descriptor is always freed, but the data block and descriptor are only freed if the reference count is one. Otherwise the reference count is decremented.

```
int freemsg(mp)
mblk_t *mp;
```

Uses successive calls to **freeb** to free all message blocks in message pointed at by **mp**.

```
mblk_t *getq(q)
queue_t *q;
```

Removes the next message from the message queue of **q** and returns a pointer to it; a nil pointer is returned if the message queue is empty. The byte count of the Queue is decremented according to the size of the message, and back-enabling may be performed. If the message queue is empty, the Queue is so marked, so that the next message enqueued will cause **q** to be scheduled.

```
int insq(q, emp, nmp)
queue_t *q;
mblk_t *emp, *nmp;
```

Places the message, **nmp**, in the message queue of **q**, immediately before the already enqueued message, **emp**. If **emp** is the nil pointer, the new message is placed at the end of the message queue. The byte count of **q** is updated and the Queue may become full.

```
int linkb(mp1, mp2)
mblk_t *mp1, mp2;
```

Concatenates the message pointed at by **mp2** at the end of the message pointed at by **mp1**.

```
int msgdsize(mp)
mblk_t *mp;
```

Returns the total number of bytes in the message blocks of **mp** which are of type M_DATA.

```
#define noenable(q) ...
```

This macro causes **q** (declared as **queue_t ***) to become unschedulable. That is, the scheduling, which is automatically performed by certain utilities, is turned off for this Queue.

```
#define OTHERQ(q)
```

This macro locates the partner of **q** (declared as **queue_t ***). That is, if **q** is the write-side Queue of the module, the pointer to the read-side Queue is returned, and vice-versa.

```
int pullupmsg(mp, len)
mblk_t mp;
int len;
```

Concatenates and aligns the first **len** bytes of message **mp** into a single message block. A **len** of -1 causes all bytes of like-type blocks to be pulled up. On successful completion a one is returned; a zero is returned on failure.

```
int putbq(q, bp)
queue_t *q;
mblk_t *bp;
```

Places the message pointed at by **bp** at the head of the message queue of **q**, in accordance with its priority (priority messages are placed at the head, regular messages are placed behind any priority messages). The same scheduling and flow control rules as in **putq** apply.

```
int putctl(q, type)
queue_t *q;
int type;
```

Creates a control message of the given type and passes it to the put procedure of the given Queue. The new message block is allocated at high priority. Returns one on successful completion, or zero if it could not allocate the proper blocks or if the given type was M_DATA, M_PROTO or M_PCPROTO.

```
int putctl1(q, type, p)
queue_t *q;
int type, p;
```

As above, but for control messages requiring a one byte parameter, p.

#define putnext(q, mp) ...

This macro calls the put procedure of the next Queue (relative to **q**) on the Stream, passing it the message pointed at by **mp**. The parameters are declared as **mblk_t *mp** and **queue_t *q**.

```
int putq(q, bp)
queue_t *q;
mblk_t *bp;
```

Places the message, **bp**, onto the message queue of Queue, **q**. Messages are enqueued according to priority. The Queue will be enabled (given to the scheduler) under the following conditions: the message is a priority message, or the queue is empty and not disabled. The Queue is marked empty when the module is first pushed, and when **getq** returns a nil pointer. The Queue becomes disabled after a call to **noenable**, and can be reenabled by calling **enableok**. The byte count is incremented and the Queue may become full.

```
int qenable(q)
queue_t *q;
```

Submits the Queue, **q**, to the STREAMS scheduler. The scheduler eventually will service the Queue by calling its service procedure.

```
int qreply(q, bp)
queue_t *q;
mblk_t *bp;
```

Send the message pointed at by **bp** in the opposite direction on the Stream, relative to the Queue **q**. This is

accomplished by locating the partner of **q** (using **OTHERQ**) and invoking its put procedure.

```
int qsize(q)
queue_t *q;
```

Returns the number of messages currently on **q**'s message queue.

```
#define RD(q) ...
```

This macro locates the read side queue_t structure given the write side Queue pointer, **q** (declared as *queue_t).

```
mblk_t *rmvb(mp, bp)
mblk_t *mp, *bp;
```

Removes block pointed at by **bp** from message pointed at by **mp**, then restores the linkage. The unlinked block is not freed. The return value is the pointer to the resulting message. If the block is not in the message, a value of -1 is returned.

```
int rmvq(q, mp)
queue_t *q;
mblk_t *mp;
```

Removes the message pointed at by **mp** from the message queue of **q**, and then restores the linkage. If the message is not on the given queue, a system panic could result.

```
int splstr()
```

Raises the processor level to block interrupts at a level appropriate for STREAMS modules executing critical

code sections. Returns the present processor level, which can later be restored by the standard **splx(s)** call.

```
int strlog(mid, sid, level, flags, fmt, arg1, ...)
short mid, sid;
char level;
ushort flags;
char *fmt;
unsigned arg1, ...;
```

Submits a message to the log driver. **mid** is the module id, **sid** is usually the minor device number. **level** is used to selective trace logged messages at a later time. **flags** specifies one or more types of logged messages (error, trace, fatal, or notify). **fmt** and **arg**'s are the same as in a **printf** call.

```
int testb(size, pri)
int size, pri;
```

Checks for the availability of a buffer of the given size, at the stated priority. Returns 1 if a buffer is available, 0 otherwise. A return of 1 does not guarantee that the next call to **allocb** will succeed.

```
mblk_t *unlinkb(mp)
mblk_t *mp;
```

Unlinks the first block of the message pointed at by mp and returns the pointer to the resulting message.

```
#define WR(q) ...
```

The opposite of **RD**. Given a pointer to the read side Queue, **q** (declared as *queue_t) returns a pointer to the write side queue.

Appendix F.

Turing Plus STREAMS Buffer Manager.

The buffer manager consists of the **mem** module, which resides in the file "mem.ch" in the **utils** subdirectory. The **streams** module is the parent of **mem**. Some repetitive and trivial or irrelevant code has been left out to preserve space, and replaced with an ellipsis (...).

file: "mem.ch"

```
parent "../streams.bd"
stub module mem
     import  (var msgb, var data)
     export (unqualified allocb,  unqualified freeb,
             unqualified freemsg, unqualified testb)
     procedure allocb(sz: nat, pri: bpri, var mp: msgb_ptr)
     procedure freeb(var bp: msgb_ptr)
     procedure freemsg(var mp: msgb_ptr)
     function testb(sz: nat, pri: bpri): boolean

     % largest and smallest buffer size; if these are changed, new
     % NBLKn constants must be added, and vice-versa; changes to either
     % will require changes also to TOTAL_BYTES calculation, and 'nblk'
     % variable initialization.
     const *MAX_POWER := 12 % expressed as power of two
     const *MIN_POWER := 2
     type  *P_RANGE : MIN_POWER..MAX_POWER % range of powers of two

     % tunable parameters
     % quantities of various buffers
     const *NBLK4096 := 0
     const *NBLK2048 := 0
     ...
     const *NBLK4    := 40

     % total number of bytes reserved for buffers
     const *TOTAL_BYTES := NBLK4096*4096+
                           NBLK2048*2048+
                           ...
                           NBLK4*4

     % threshold values for denying LO and MED priority requests
     % expressed as percentages
     const *STRLOFRAC  := 0.7
     const *STRMEDFRAC := 0.85

end mem
```

```
body module mem

    const *MAX_BUFFER_SIZE := 2**MAX_POWER % MAX expressed as integer
    const *MIN_BUFFER_SIZE := 2**MIN_POWER

    % super BUFFER containing all data buffers
   var BUFFERS : array 1..TOTAL_BYTES of int1

    % array of freelist pointers for various buffer sizes
    var freelist : array P_RANGE of addressint

    % counters of how many buffers of each size are used
    var inuse : array P_RANGE of nat

    % low and medium priority thresholds for different buffer sizes
    var lo_thresh, med_thresh : array P_RANGE of nat

    % this array holds the values of NBLKn used to simplify loops
    var nblk : array P_RANGE of nat :=
              init(NBLK4, NBLK8, NBLK16, NBLK32, NBLK64, NBLK128,
                   NBLK256, NBLK512, NBLK1024, NBLK2048, NBLK4096)

    function sizetopower(sz: nat): 0..MAX_POWER
    % this internal function converts a buffer size in bytes into
    % a suitable buffer size, expressed as a power of two or returns
    % zero if an invalid size is given

     ...

    end sizetopower

    procedure allocbuff(pwr: P_RANGE, pri: bpri, var b: addressint)
    % internal procedure to allocate buffer of size 2**pwr bytes

        if (pwr = 0) then % this would mean that the size passed to
           b := nilAddr % allocb was out of range (see sizetopower)
        else
           if ((freelist(pwr) = nilAddr) or
             ((pri=bpri.BPRI_LO) and (inuse(pwr)>lo_thresh(pwr))) or
             ((pri=bpri.BPRI_MED) and (inuse(pwr)>med_thresh(pwr))))
            then
               b := nilAddr % cannot allocate because of depletion
           else
               inuse(pwr) += 1
               b := freelist(pwr)
               freelist(pwr) := addressint@(b)
           end if
        end if

    end allocbuff
```

```
body procedure allocb % (sz: nat, pri: bpri, var mp: msgb_ptr)

     var dp : datab_ptr
     var bp : addressint
     var pwr := sizetopower(sz)

     allocbuff(pwr, pri, bp)
     if (bp = nilAddr) then % could not allocate buffer
          mp := nilMsgb
     else % buffer allocated--try to allocate other pieces
          new msgb, mp
          new datab, dp
          if ((mp ~= nilMsgb) and (dp ~= nilDatab)) then
               % mp and dp allocated--can proceed
               bind var p to msgb(mp)
               p.b_next   := nilMsgb % initialize various fields
               p.b_prev   := nilMsgb
               ...

               bind var b to datab(dp)
               b.db_freep := nilDatab
               b.db_base  := bp
               ...

          else
               % could not allocate mp or dp or both--clean up
               addressint@(bp) := freelist(pwr)
               freelist(pwr) := bp
               inuse(pwr) -= 1
               if (mp ~= nilMsgb) then
                    free msgb, mp
               end if
               if (dp ~= nilDatab) then
                    free datab, dp
               end if
          end if
     end if

end allocb

body function testb % (sz: nat, pri: bpri) : boolean

     var pwr := sizetopower(sz)
     if (pwr = 0) then % size was out-of-bounds
        result false
     else
        result
        ~((freelist(pwr) = nilAddr) or
        (pri=bpri.BPRI_LO) and (inuse(pwr)>lo_thresh(pwr))) or
        (pri=bpri.BPRI_MED) and (inuse(pwr)>med_thresh(pwr))))
     end if

end testb
```

```
    body procedure freeb % (var bp : msgb_ptr)

        if (bp ~= nilMsgb) then
            ...  % various assert statements
            var dp := msgb(bp).b_datap
            var b := datab(db).db_base

            free msgb, bp
            if (datab(dp).db_ref > 1) then
                datab(dp).db_ref -= 1)
            else
                var p := sizetopower(datab(dp).db_lim - b)
                addressint@(b) := freelist(p)
                freelist(p) := b
                inuse(p) -= 1
                free datab, dp
            end if
        end if

    end freeb

    body procedure freemsg % (var mp : msgb_ptr)

        loop
            exit when (mp = nilMsgb)
            var nmp := msgb(mp).b_cont
            freeb(mp)
            mp := nmp
        end loop

    end freemsg

    % initialization
    var nb := address(BUFFERS)
    for i : P_RANGE
        lo_thresh(i) := round(STRLOFRAC * nblk(i))
        med_thresh(i) := round(STRMEDFRAC * nblk(i))
        if (nblk(i) > 0) then
            freelist(i) := nb
            for j : 1..nblk(i)-1
                var b := nb
                nb += 2**i
                addressint@(b) := nb
            end for
            addressint@(nb) := nilAddr
            nb += 2**i
        else
            freelist(i) := nilAddr
        end if
        inuse(i) := 0
    end for

end mem
```

Appendix G.

Turing Plus STREAMS Scheduler.

The scheduler consists of two parts: the module **sched** and the monitor **sched_mon.** The two are in separate files, with the monitor being a child of the module.

file: "sched.ch"

```
parent "../streams.bd"
stub module sched

     import (var qinit, var queue)
     export (unqualified qenable)
     child "sched_mon.ch"

end sched

body module sched

     grant (var queue)
     child "sched_mon.ch"

     process sched_proc

          var q : queue_ptr
          loop
               schedmon.dequeue(q)
               qinit(queue(q).q_qinfo).qi_srvp(q)
          end loop

     end sched_proc

     fork sched_proc

end sched
```

```
file: "sched_mon.ch"

parent "sched.ch"
stub monitor schedmon

     import (var queue)
     export (unqualified qenable, dequeue)

     procedure qenable (q : queue_ptr)
     procedure dequeue (var q : queue_ptr)

end schedmon

body monitor schedmon

     type q_q : record
                    head, tail : queue_ptr % a queue of Queues
                end record

     var ready : q_q
     var more_ready : condition % signalled when ready queue not empty

     function is_in_q(q : queue_ptr, qq : q_q) : boolean
     % this function checks whether q is in qq

          var q1: queue_ptr := qq.head
          loop
               exit when ((q1 = q) or (q1 = nilQueue))
               q1 := queue(q1).q_link
          end loop
          result (q1 = q)

     end is_in_q

     body procedure qenable % (q: queue_ptr)
     % this procedure schedules a Queue for execution

          assert (q ~= nilQueue) % sanity check
          % make sure q is not already scheduled
          assert (~is_in_q(q, ready) )

          if (ready.head = nilQueue) then
               ready.head := q
          else
               queue(ready.tail).q_link := q
          end if
          ready.tail := q
          queue(q).q_link := nilQueue
          % signal, in case scheduler is waiting
          assert (ready.head ~= nilQueue)
          signal more_ready
     end qenable
```

```
  body procedure dequeue % (var q: queue_ptr)
  % this procedure gets the next Queue to schedule

       if (ready.head = nilQueue) then
            wait more_ready
       end if
       assert (ready.head ~= nilQueue)
       q := ready.head
       ready.head := queue(q).q_link
       if (ready.head = nilQueue) then
            ready.tail := nilQueue
       end if

  end dequeue

  % initialization
  ready.head := nilQueue
  ready.tail := nilQueue

end schedmon
```

Appendix H.

Class I LLC Protocol Module.

The relevant code is contained in three files, all of which reside in the **drivers** subdirectory. The file "llc1_types.in" is included by the **streams** module and contains the necessary type and constant declarations. The other two files are the stub and body, respectively, of the **llc1** module, which is a child of the **streams** module. Some repetitive and trivial or irrelevant code has been left out to preserve space, and replaced with an ellipsis (...).

```
file: "llc1_types.in"

% address types
type  * stn_addr : nat2 % station addresses, Ethernet addresses are 16
                        % or 48 bits, we choose 16, but that can easily
                        % be changed
type  * SAP_addr : nat1 % SAP addresses
const * CR_bit := 2#00000001 % Command/Response bit position
const * CR_mask := CR_bit xor 2#11111111
const * IG_bit := 2#00000001 % Individual/Group bit position
const * NULL_SAP := 0
const * GLOBAL_DA := 2#11111111

% control byte formats
const * UI := 2#00000001
const * XID := 2#10101111
const * TEST := 2#11100011
const * PF_bit := 2#00010000
const * PF_mask := PF_bit xor 2#11111111

% transmission status values
const * XMIT_OK := 255
const * XMIT_FAIL := 254

% request primitives
const * XID_req := XID
const * TEST_req := TEST
const * ACTIVATE_req := 253
const * DEACTIVATE_req := 252
const * UP_req := 251
const * DOWN_req := 250

% component states
const * ACTIVE := ACTIVATE_req
const * INACTIVE := DEACTIVATE_req
const * UP := UP_req
const * DOWN := DOWN_req

% header format
type  * ctl_field : nat1 % for Class 1 LLC this is enough
type  * pri_field : nat1 % one byte should be enough for priority
type  * scl_field : nat1 % same for service class

type  * llc1_header :
/* The actual LLC header consists of the source and destination SAP
addresses, the control field, and the priority and service class fields.
The DA/SA addresses and the length field make up the MAC header.  They
are included because there is no other way of passing parameters in a
message-based system (such as STREAMS).  Passing them in a separate
message block would serve little purpose other than degrading
performance.  */
```

```
        record
           DA, SA : stn_addr
           len : nat1
           DSAP, SSAP: SAP_addr
           control : ctl_field
           pri : pri_field
           scl : scl_field
        end record

   const * min_header_size :=
          size(llc1_header) - size(pri_field) - size(scl_field)


file "llc1.st"

     parent "../streams.bd"
     stub module llc1
        import(var u,
               var incoreInode, major, minor, var cdevsw,
               var fmodsw,
               include "../allocators/a_grant.in" ,
               include "../utils/u_grant.in"
              )
        export(c_init)
        function c_init : streamtab_ptr
     end llc1
```

```
file: "llc1.bd"

body "llc1.st" module llc1

    /************************************
     * variables for STREAMS structures *
     ************************************/
     var llc1info          : module_info_ptr
     var rsinfo, wsinfo    : module_stat_ptr
     var urinit, uwinit,
         lrinit, lwinit    : qinit_ptr
     var st                : streamtab_ptr

    /************************************
     * private per SAP data structures *
     ************************************/
     var SAPs : collection of forward SAP_rec
     type SAP_rec : record
                      next, prev : pointer to SAPs
                      adr : SAP_addr
                      qptr : queue_ptr
                      state : INACTIVE..ACTIVE % SAP component state
                    end record
     var SAP_list := nil(SAPs) % a list of currently opened SAP's

    /*********************************************
     * private station component data structures *
     *********************************************/
     var state : DOWN..UP := DOWN % station  component state
     var llc1bot : queue_ptr := nilQueue % linked lower queue
     var nextSAP := SAP_list % next upper queue to be serviced

    /*******************
     * llc1 procedures *
     *******************/

     function next_sap : nat1
     /* Find next available SAP address.  Search starts at 252 and
        goes down by four, until four, since bits 0 and 1 are reserved,
        and address 0 is the controlling stream.
        A zero is returned when no more SAP addresses are available.
     */
```

```
   var register i := 252
   loop
      var register s := SAP_list
      loop
         exit when ((s = nil(SAPs)) or (SAPs(s).adr = i))
         s := SAPs(s).next
      end loop
      if (s = nil(SAPs)) then
         exit
      end if
      i -= 4
      exit when (i = 0)
   end loop
   result i
end next_sap

procedure send(var mp : msgb_ptr)
/* Sends the given message down the lower linked stream.
   If a group DSAP is given, then the GLOBAL DA is used. */

   bind var hdr to llc1_header@(msgb(mp).b_rptr)
   if ((hdr.DSAP and IG_bit) ~= 0) then % group DSAP
      hdr.DA := GLOBAL_DA               % must send to all stations
   end if
   putnext(llc1bot, mp)
end send

procedure llc1open(q : queue_ptr, dev : dev_t,
                   flag : openFlag, sflag : strmFlag,
                   var retval : int)

   var mdev : nat1
   if (sflag = strmFlag.CLONEOPEN) then
      if (SAP_list = nil(SAPs)) then
         mdev := 0
      else
         mdev := next_sap
      end if
   else
      mdev := minor(dev)
   end if
   if (SAP_list  = nil(SAPs) and mdev  > 0) or
      (SAP_list ~= nil(SAPs) and mdev <= 0) then
      retval := OPENFAIL
   else
```

```
      var n : pointer to SAPs
      new SAPs, n
      SAPs(n).next := SAP_list
      if (SAP_list ~= nil(SAPs)) then
         SAPs(SAP_list).prev := n
      end if
      SAP_list := n
      SAPs(n).qptr := q
      queue(q).q_ptr := addr(SAPs(n))
      SAPs(n).adr := mdev
      SAPs(n).state := ACTIVE
   end if
end llc1open

procedure llc1close(q : queue_ptr, flag : closeFlag)
/*
 * Upper queue close.
 */
   var sap := queue(q).q_ptr % address of private SAP record
   var p := SAP_list % pointer for searching
   loop % check that SAP address is valid (i.e. it is in SAP_list)
      exit when ((p = nil(SAPs)) or (addr(SAPs(p)) = sap))
      p := SAPs(p).next
   end loop
   if (p ~= nil(SAPs)) then
      var prev := SAPs(p).prev
      var next := SAPs(p).next
      if (prev = nil(SAPs)) then
         SAP_list := next
      else
         SAPs(prev).next := next
      end if
      if (next ~= nil(SAPs)) then
         SAPs(next).prev := prev
      end if
      if (SAPs(p).adr = NULL_SAP) then % closing control stream
         state := DOWN  % put station into DOWN state
      end if
      free SAPs, p
   else
      put "llc1 close: attempt to close non-existent SAP\n"
   end if
end llc1close
```

```
procedure llcluwput(q : queue_ptr, var mp : msgb_ptr)
/* This procedure processes all messages relevant to the SAP
   component, such as the requests to activate and deactivate.
   Data messages, and requests to send a TEST or XID are enqueued
   provided that the SAP component is active, otherwise they are
   discarded.  Requests to change the state of the station
   component are discarded unless they come from the controlling
   stream.  IOCTL messages are handle here, even though they
   pertain to the station component.  This is because the IOCTL
   message contains the link block which carries the bottom linked
   queue pointer, which is needed before any message is handled by
   the (station component side) lower service procedure.  Generic
   head processing is done on FLUSH messages.  All other message
   types are discarded.
*/

   type status : enum(FREE, ENQUEUE, NAK)
   var s : status := status.FREE

   case (datab(msgb(mp).b_datap).db_type) of

      label m.M_FLUSH:
         bind var ftype to mtset@(msgb(mp).b_rptr)
         if (mt.FLUSHW in ftype) then
            flushq(q, mt.FLUSHDATA)
         end if
         if (mt.FLUSHR in ftype) then
            flushq(RD(q), mt.FLUSHDATA)
            ftype -= mtset(mt.FLUSHW)
            qreply(q, mp)
            return
         end if

      label m.M_IOCTL: /* Only controlling stream can do ioctl's.
                          Two calls are recognized: LINK and UNLINK
                          */
         s := status.NAK
         bind var sap to SAP_rec@(queue(q).q_ptr)
         if (sap.adr = NULL_SAP) then
            bind var iocp to iocblk@(msgb(mp).b_rptr)
            case (iocp.ioc_cmd) of
               label ioc_c.I_LINK:
                  if (llc1bot ~= nilQueue) then
                     s := status.NAK
                  else
                     bind var linkp to
                        linkblk@(msgb(msgb(mp).b_cont).b_rptr)
                     llc1bot := WR(linkp.l_qbot)
                     datab(msgb(mp).b_datap).db_type := m.M_IOCACK
                     iocp.ioc_count := 0
                     qreply(q, mp)
                     return
                  end if
```

```
                label ioc_c.I_UNLINK:
                   bind var linkp to
                        linkblk@(msgb(msgb(mp).b_cont).b_rptr)
                   llc1bot := nilQueue
                   datab(msgb(mp).b_datap).db_type := m.M_IOCACK
                   iocp.ioc_count := 0
                   qreply(q, mp)
                   return
             end case
          end if

       label m.M_PCPROTO:
          if ((msgb(mp).b_wptr-msgb(mp).b_rptr) >= min_header_size)
          then
             bind var hdr to llc1_header@(msgb(mp).b_rptr)
             bind var sap to SAP_rec@(queue(q).q_ptr)
             case (hdr.control) of
                label ACTIVATE_req, DEACTIVATE_req:
                   sap.state := hdr.control
                label UP_req, DOWN_req:
                   if (sap.adr = NULL_SAP) then
                      s := status.ENQUEUE
                   end if
                label XID_req, TEST_req:
                   if (sap.state = ACTIVE) then
                      s := status.ENQUEUE
                   end if
             end case
          end if

       label m.M_PROTO:
          if ((msgb(mp).b_wptr-msgb(mp).b_rptr) >= min_header_size)
          then
             bind var sap to SAP_rec@(queue(q).q_ptr)
             if (sap.state = ACTIVE) then
                bind var hdr to llc1_header@(msgb(mp).b_rptr)
                hdr.control := UI
                s := status.ENQUEUE
             end if
          end if

    end case

    if (s = status.FREE) then
       freemsg(mp)
    elsif (s = status.NAK) then % fail ioctl
       datab(msgb(mp).b_datap).db_type := m.M_IOCNAK
       qreply(q, mp)
    else % (s = status.ENQUEUE)
```

```
      if (llc1bot = nilQueue) then % no bottom queue - ERROR
         datab(msgb(mp).b_datap).db_type := m.M_ERROR
         msgb(mp).b_rptr := datab(msgb(mp).b_datap).db_base
         nat1@(msgb(mp).b_rptr) := EINVAL
         msgb(mp).b_wptr := msgb(mp).b_rptr + 1
         qreply(q, mp)
      else
         putq(q, mp)
         qenable(llc1bot)
      end if
   end if

end llc1uwput

procedure get_next_q(var nq : queue_ptr)
/* Round-robin scheduling.
   Return next upper queue that needs servicing.
   Returns nilQueue when no more work needs to be done.
*/

     var register i := nextSAP
     var found := false
     loop
        nq := SAPs(i).qptr
        i := SAPs(i).next
        if (i = nil(SAPs)) then
           i := SAP_list % wrap around
        end if
        if (nq ~= nilQueue) then
           nq := WR(nq)
           if (queue(nq).q_first ~= nilMsgb) then
              found := true
           end if
        end if
        if (i = nextSAP) then
           exit /* went all the way around */
        end if
     end loop

     nextSAP := i; /* round robin */

     if (not found) then
        nq := nilQueue % no more work to be done
     end if

end get_next_q
```

```
procedure llc1lwsrv(q : queue_ptr) % called with (q == llc1bot)
/* This procedure performs the actions of the station component.
   It sends UI, TEST, and XID messages on behalf of the SAP
   components and processes requests to put the station component
   into an UP or DOWN state. As all service procedures, it is a
   loop which ends only when no more work needs to be done (as
   indicated by the return of a nil pointer from get_next_q) or
   the way down is blocked due to flow control.
*/
   loop
      exit when (not canput(queue(q).q_next)) % blocked below
      var nq : queue_ptr
      get_next_q(nq)
      exit when (nq = nilQueue) % no more work -- so long
      var mp : msgb_ptr
      getq(nq, mp)
      case (datab(msgb(mp).b_datap).db_type) of

         label m.M_PROTO: % this must be a UI request - send it
            send(mp)
            return

         label m.M_PCPROTO: /* Can only be TEST or XID req. from
                               active SAP or UP/DOWN request. The
                               latter must be from the controlling
                               stream.
                            */
            bind var hdr to llc1_header@(msgb(mp).b_rptr)
            bind var sap to SAP_rec@(queue(q).q_ptr)
            case (hdr.control) of
               label UP_req, DOWN_req:
                  if (sap.adr = NULL_SAP) then
                     state := hdr.control
                  end if
               label XID_req, TEST_req:
                  datab(msgb(mp).b_datap).db_type := m.M_PROTO
                  hdr.SSAP and= CR_mask % send as a command
                  send(mp)
                  return
            end case

            freemsg(mp)

      end case
   end loop

end llc1lwsrv

function find_sap(hdr : llc1_header) : pointer to SAPs
/* This function finds the pointer to the sap record of the SAP
   whose address matches the DSAP of the header.
*/
```

```
   var register dsap := hdr.DSAP
   var register sap := SAP_list
   loop
      exit when (sap = nil(SAPs)) or (SAPs(sap).adr = dsap)
      sap := SAPs(sap).next
   end loop
   result sap
end find_sap

function find_group(hdr : llc1_header) : pointer to SAPs
/* This functions takes the group DSAP address in the header and
   returns the pointer to the first SAP address which belongs to
   the group.  The rest of the SAP's are linked via the next field
   of SAP_rec.  At present, groups are not implemented, so the
   function simply returns a nil pointer - signifying that the
   group is empty.  Any future implementation of groups should
   follow this convention of linking all the SAP_records belonging
   to the same group.  The rest of the implementation should not
   matter.
*/
   result nil(SAPs)

end find_group

procedure llc1lrput(q : queue_ptr, var mp : msgb_ptr)
/* This procedure routed M_PROTO messages up to the appropriate
   SAP, provided the SAP component is ACTIVE and the P/F bit is
   off.  All other valid message types are enqueued and processed
   by the lower read service procedure.  Valid messages are
   M_PROTO, M_PCPROTO and M_FLUSH.  No other messages are
   anticipated, so all others are simply discarded.
 */

   case ( datab(msgb(mp).b_datap).db_type ) of

      label m.M_PROTO, m.M_PCPROTO:
         var group: boolean
         if ((msgb(mp).b_wptr-msgb(mp).b_rptr) < min_header_size)
       then
            freemsg(mp) % header not correct - discard
         else % header is all right
            bind var hdr to llc1_header@(msgb(mp).b_rptr)
            if (hdr.DSAP = NULL_SAP) then
               putq(q, mp) % addressed to station component
            else % addressed to individual or group SAP
               var sap : pointer to SAPs
               group := ((hdr.DSAP and IG_bit) ~= 0)
               if (group) then
                  sap := find_group(hdr)
               else
                  sap := find_sap(hdr)
               end if
```

```
            loop
               if (group and (sap = nil(SAPs))) then
                  exit % no one here from this group
               end if
               if (sap = nil(SAPs)) then
                  put "llcllrput: message addressed to ",
                      "non-existent SAP\n"
               else % SAP exists
                  if (SAPs(sap).state = ACTIVE) then
                     var ok := true
                     case (hdr.control) of
                        label UI:
                           if ((hdr.SSAP and CR_bit) ~= 0) or
                              ((hdr.control and PF_bit) ~= 0)
                           then
                              ok := false % a UI response or UI
                                          % with PF-bit on is
                           end if         % discarded
                        label XID, TEST:
                        label: % default - discard
                           ok := false
                     end case
                     var uq := SAPs(sap).qptr
                     if (ok and canput(nextsq(q))) then
                        if (group) then
                            var mc : msgb_ptr % copy of message
                            dupmsg(mp, mc)
                            putnext(q, mc)
                        else
                            putnext(q, mp)
                        end if
                     end if
                  end if % SAP active
               end if % SAP doesn't exists
               if not(group) then
                  exit % individual DSAP - no more work here
               else
                  sap := SAPs(sap).next % else do next in group
               end if
            end loop
         end if % NULL_DSAP address
      end if % header incorrect

      if (group) then
         freemsg(mp) % free original message
      end if

   label m.M_FLUSH:
      putq(q, mp) % let the service procedure handle FLUSH
```

```
      label: % default - everything else is discarded
         freemsg(mp)

   end case

end llc1lrput

procedure llc1lrsrv(q : queue_ptr)
/* This procedure handles everything passed-up by the lower read
   put procedure, namely NULL_DSAP'ed M_PROTO and M_PCPROTO
   messages, FLUSH messages.  The former will include TEST and XID
   commands and responses (in M_PROTO messages) and transmission
   status reports (in M_PCPROTO messages).  Anything not covered by
   the above will be discarded.
*/
   loop
      var mp : msgb_ptr
      getq(q, mp)
      exit when (mp = nilMsgb)
      type status : enum(FREE, DONT_FREE)
      var s : status := status.FREE
      case (datab(msgb(mp).b_datap).db_type) of

         label m.M_FLUSH: % generic stream-head FLUSH processing
            bind var ftype to mtset@(msgb(mp).b_rptr)
            if (mt.FLUSHR in ftype) then
               flushq(q, mt.FLUSHALL)
            end if
            if (mt.FLUSHW in ftype) then
               ftype -= mtset(mt.FLUSHR)
               qreply(q, mp)
               s := status.DONT_FREE
            end if

         label m.M_PCPROTO:
            bind var hdr to llc1_header@(msgb(mp).b_rptr)
            case (hdr.control) of
               label XMIT_OK:    /* transmission status messages */
               label XMIT_FAIL:  /* are currently ignored */
                end case

         label m.M_PROTO: /* Only NULL_DSAP messages should get
                             here, so the only types to expect are
                             XID and TEST command and responses.
                            */
            bind var hdr to llc1_header@(msgb(mp).b_rptr)
            case (hdr.control) of
```

```
                label TEST, XID:
                   if ((hdr.SSAP and CR_bit) = 0) then % command
                      var tdsap : SAP_addr := hdr.DSAP % reverse
                                                       % addresses
                      var tda : stn_addr := hdr.DA
                      hdr.DSAP := hdr.SSAP
                      hdr.SSAP := tdsap
                      hdr.DA := hdr.SA
                      hdr.SA := tda
                      hdr.SSAP or= CR_bit % change to response
                      send(mp) % and send back
                      s := status.DONT_FREE
                   end if
             end case

      end case
      if (s = status.FREE) then
         freemsg(mp)
      end if
   end loop

end llc1lrsrv

body function c_init
    result st
end c_init

% module information structure
mi_init(llc1info, stm.llc1, "llc1", 0, INFPSZ, 0, 0)

% read module statistics structure
ms_init(rsinfo, 0, 0, 0, 0, 0, 0, 0)

% write module statistics structure
ms_init(wsinfo, 0, 0, 0, 0, 0, 0, 0)

% upper read queue initialization structure
qi_init(urinit, nil_qi_putp, nil_qi_srvp, llc1open,
        llc1close, nil_qi_admin, llc1info, rsinfo)

% upper write queue initialization structure
qi_init(uwinit, llc1uwput, nil_qi_srvp, nil_qi_open,
        nil_qi_close, nil_qi_admin, llc1info, rsinfo)

% lower read queue initialization structure
qi_init(lrinit, llc1lrput, llc1lrsrv, nil_qi_open,
        nil_qi_close, nil_qi_admin, llc1info, wsinfo)

% lower write queue initialization structure
qi_init(lwinit, nil_qi_putp, llc1lwsrv, nil_qi_open,
        nil_qi_close, nil_qi_admin, llc1info, wsinfo)
```

```
        % streamtab initialization structure
        st_init(st, urinit, uwinit, lrinit, lwinit)

        % c_init will be called to initialize the element of cdevsw

end llcl
```