PACKET RADIO SIMULATION AND PROTOCOLS

by

Daniel R. Helman
S.B., S.M., Massachusetts Institute of Technology
(1980)

SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS OF THE DEGREE OF

DOCTOR OF PHILOSOPHY
IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
June 1986

Signature of Author_____
Department of Electrical Engineering and Computer Science
June 24, 1986

Certified by_____
Robert G. Gallager
Thesis Supervisor

Accepted by_____
Arthur C. Smith
Graduate Officer

PACKET RADIO SIMULATION AND PROTOCOLS

by

Daniel R. Helman

Submitted to the Department of Electrical Engineering and Computer Science on June 25, 1986 in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Electrical Engineering and Computer Science

ABSTRACT

A system for simulating packet data networks is presented. The system is designed to compare the performance of competing communication algorithms. Its architecture features two levels of multiprocessing. First, the network nodes consist of multiple, homogeneous processors with no master processor. Second, the nodes are interconnected via a unique high bandwidth digital switch. The switch is capable of forming both wire and radio (i.e. multiaccess) connections.

Simulation experiments are programmed using a special communication oriented programming language and operating system. Programs execute cooperatively in each node, without a master processor.

Together the hardware and software can emulate the characteristics (baud rate, error rate, contention, etc.) of a broad range of wire and radio networks. Packet radio protocols are discussed. A new protocol with superior performance is proposed.

Thesis Supervisor: Professor Robert G. Gallager

Keywords: Simulation, Packet Radio, Multiprocessing, Distributed Processing

ACKNOWLEDGEMENTS

Many people have contributed directly and indirectly to the (CLUMPS) project. I would like to thank my thesis supervisor, Professor Robert Gallager, for his tremendous help when I needed it and independence when I wanted it. The four other students on the project, Mark Tubinis, Shahrukh Merchant, Alberto Sentieri-Filho and Georgios Nikolaou, all made significant contributions. The NIL group at IBM in Yorktown, especially John Pershing, taught us the right way to think about computer languages (although sometimes I wished they hadn't).

Several government and corporate sponsers helped us with grants and equipment. Many thanks go to the NSF (contract ECS-8310698) for supporting me as a research assistant, and to IBM for fellowship support. We are also very grateful to IBM, Motorola and Pro-Log for their generous equipment donations.

Most of all, though, I am indebted to all of the above people, and especially my wife, Carol, for their extreme patience during a two year project that took four and a half years.

Table of Contents

Chapter 1

INTRODUCTION

Most complex systems are not completely amenable to analysis, and thus simulation becomes a necessary design tool. Digital data communication networks fit into this category. We will develop a system capable of simulating a wide variety of these networks. Our primary interest, however, is using it to explore packet radio network protocols. Although our final goal is to create improved protocols, we will concentrate here on the simulation system and its design.

First we develop models for packet communication (both wire and radio). The models clearly define the networks to be simulated. One problem is that any model will either depart from actual systems or lump different systems together and treat them as identical. These limitations, trade-offs and simplifications inherent in the models are discussed.

The system we have designed to perform simulations has been called (CLUMPS) for Communication Language Multiprocessor System. It is a collection of special purpose hardware and software designed to mimic a wide variety of communication networks. It consists of a number of network nodes (in one to one correspondence with the network being simulated) and a special computer language. The hardware provides flexibility of interconnection and link characteristics. The software provides flexibility of protocols, packet generation and measurement. An important feature is that the software can be written independently of the hardware configuration.

(CLUMPS) was designed for low cost as well as faithful emulation of the models. The design uses many identical processor boards, each with a

simple 8-bit microprocessor. Where practical, software is substituted for hardware. Our underlying concept is that we can simulate any network by copying its structure, but we may have to scale down its transmission rates. For example, to simulate Ethernet[1] with this hardware would require data rates much less than 10 megabits/sec, but if every speed parameter is scaled accordingly the simulation can still be very accurate.

A modular programming language called Network Implementation Language[2] is chosen for writing simulation experiments. It has features such as queues and message passing that are designed to simplify programming of communication tasks. Its modularity and multitasking capabilities allow the experimenter to combine standard protocol pieces from libraries with experimental ones.

We discuss the limitations of this design, and alternatives. The hardware and software are described in detail. The most unique feature of the system is that node interconnection is accomplished with a digital switch that is capable of arbitrary multiaccess connections. Another unique feature is the architecture of the network nodes in which there is no controlling processor. Rather, they have multiple, homogenous processors which cooperate on the computation and communication tasks.

Finally, some problems of packet radio communication are introduced. We discuss protocols for radio communication at the data link layer. A new protocol is proposed for testing on the (CLUMPS) system.

---

1. R. M. Metcalfe and D. R. Boggs, "Ethernet: Distributed Packet Switching for Local Computer Networks," Communications of the ACM, vol. 19, pg. 395, July 1976

2. Draft NIL Reference Manual, RC 9732, IBM T. J. Watson Research Center, Yorktown Heights, NY, Dec. 1982

1.1 Network Communication Model

Since we cannot simulate complete communication networks in ALL their complexity, we design models that have the networks' key parameters. The models are simply described with only a few parameters. Our system can then simulate the model exactly.

A data network consists of two elements: nodes and links. The nodes are computers that generate, forward, and digest messages to/from other nodes. The links are communication channels which connect two or more nodes. Each channel is capable of carrying one message at any given time, independent of all others.

Wire networks are commonly represented as undirected graphs, since nodes are usually connected by two channels, one in each direction, with similar characteristics. In our system all links are bidirectional. Each link is defined by three physical parameters: bit rate, error rate, and delay. Bit rate is the number of bits per second that can be sent. Error rate is the percentage of bits that are received in error. Actually this number would be better characterized by a time varying distribution of errors, since some links have the property of errors coming in groups, and often the quality of a connection will change over time. Delay is the time between when a bit is sent, and when it is received.

Often networks are modelled at a higher level, assuming a low level protocol that causes data received in error to be retransmitted until it is correctly received. In this case there is no error rate, but the delay is variable, with a distribution based on the low level parameters and protocol.

1.2 Radio Communication Model

Radio networks add another level of complexity. They provide broadcast links that can connect many nodes at once. Each node converses with whomever is physically close enough to be in range. The set of nodes that are in range can be different for every node on the broadcast link. The nodes in range of another node are called its neighbors.

In reality a radio may be equipped with a very sensitive receiver, but a weak transmitter, or vice-versa, making some connections unidirectional. This is undesirable, however, due to the need to acknowledge correctly received transmissions. If one's transmitter were more (or less) capable than one's neighbor's, it would mean the link would be one-way, and could not be used for acknowledgement messages.

Communication over broadcast links is different from wire links in three ways. First, we cannot direct a transmission at a particular receiver; all transmissions go to every receiver within range. Sometimes we will deal with several frequencies which are modelled as distinct broadcast links operating concurrently. If all frequencies have the same range, the result could also be modelled by time multiplexing a single frequency.

Second, if two or more of a given node's neighbors transmit at the same time, the packets will overlap and the given node will be unable to receive either. This is called a collision. For all radios in range, a collision (or noisy reception) results in the knowledge that an error occurred, but not any other information. This information is obtained when the channel goes idle, and the previous reception can be checked.

Third, during transmission, a radio's receiver is so swamped by its transmitter that it will not receive anything else on that frequency. In other words no reception is possible during transmission. This eliminates schemes to back off if there is a collision during transmission, such as

in Ethernet.

To illustrate these concepts, suppose five nodes all equipped with identical single frequency radio transceivers are arranged in a line such that each node has only one or two neighbors:

A ------ B ------ C ------ D ------ E

If B and D transmit at the same time C will know that a collision occurred, but A will receive B's message, and E will receive D's message. However, A and D can safely transmit at the same time.

As with wire networks, radio links can be characterized by bit rate, error rate and delay. The error rate pertains to errors caused by a noisy channel, not from collisions. Delay comes from the distance between the transmitter and receiver. This means that there can be a different delay for each pair of nodes connected to a broadcast link.

Collisions can come from one of two sources. First, two nodes in range of each other could simultaneously decide to transmit. These will be called direct collisions. Second, two nodes, not in range of each other, could decide to send messages to the same receiver. These are indirect collisions. Most packet radio systems eliminate direct collisions with a method called carrier sensing. Nodes wait until none of their neighbors are transmitting before they start to transmit.

A key parameter in a carrier sense system is the delay from the time that a node makes the decision to begin transmission to the time when a receiving node detects the start of that transmission. During this period conflicting nodes may both decide to start a transmission. Obviously, the longer this delay, the greater is the problem of direct conflict. Notice that this delay includes the node to node transmission delay, as well as some internal processing delay. It will typically be much smaller than the length of time it takes to transmit a packet. For example, two nodes 20 miles apart communicating at 50 Kbit/sec suffer a transmission delay of less than one byte.

Indirect collisions, however, are a more serious problem, for which

there is no simple solution. Receivers are vulnerable to an indirect collision during the entire reception, not just for a short period at the beginning. If the protocol being used requires the receiver to immediately acknowledge a successful reception, then indirect collisions can be caused by the transmitter's neighbors as well as the receiver's.

To summarize, the model of a radio network consists of an undirected graph, where the nodes correspond to radios, and the links connect a radio with all others in range. A radio is either transmitting, receiving or idle. If a node is transmitting none of its neighbors are idle. When a node goes from receiving to idle it either receives a valid message, or knows that an error occurred. Errors can be due to noise or collisions. Bit rate, error rate, and a limited form of delay are model parameters.

Although (CLUMPS) can emulate more general packet radio systems, all protocols discussed here will use carrier sensing, that is, will only start to transmit from the idle state. This means that all state transitions occur between idle and some other state.

Chapter 2

(CLUMPS)

2.1 Goals

The CLUMPS (Communication LangUage MultiProcessor System) project was started to create a system for testing packet communication protocols with high accuracy and low cost. The goals were as follows:

1. The system should be flexible enough to simulate many types of networks, including packet radio.

2. The system should be as faithful to the communication model as possible. Links should be independent. Computation overhead should be low. Performance should be based on communication parameters and choice of protocol only.

3. The cost per node should be about $1000, and the system should be capable of at least 32 nodes.

4. Experiments should be programmed in a high-level language to make it easy to make changes and try different ideas. The language should be modular, so that features of a protocol that are not critical to the current experiment (e.g. routing) can be drawn from a library without recoding. This will also make the system easier to use for someone who is interested in simulation results, not method.

2.2 Structure

(CLUMPS) was designed with multiple computer nodes and communication links, to run in real time and to simulate protocols by actually using them to pass packets around a network. The nodes have up to eight bidirectional communication lines, each with its own controlling processor. This was done to insure independence of the nodes, to increase the modularity of the system, and to experiment with multiprocessing. A node's processors exchange information within the node via a shared memory. The serial communication lines use standard asynchronous protocol, with baud rates up to 76.8 Kbits/sec. Links are connected together to form a network through a digital switch. The switch allows any network configuration to be selected under software control, including radio-type connections. See figure 1.

The most unusual aspect of the design is the absence of a master processor in each node. All programs are executed in a distributed fashion by the eight homogeneous communication processors. Programs reside in the shared memory, and consist of many tasks that are shared by the processors. At any given time a processor can be executing one of these tasks in addition to its duties in servicing its communication line.

2.2.1 Model Limitations

While the system comes very close to implementing the communication models discussed in the previous section, it does fall short in a few areas, the most significant of which is delay.

Delay is difficult to simulate exactly. One problem is that the delay added by the hardware and software in processing a packet is impossible to access. In the radio environment delay should be dependent on the physical distance between the transmitter and receiver. This is not

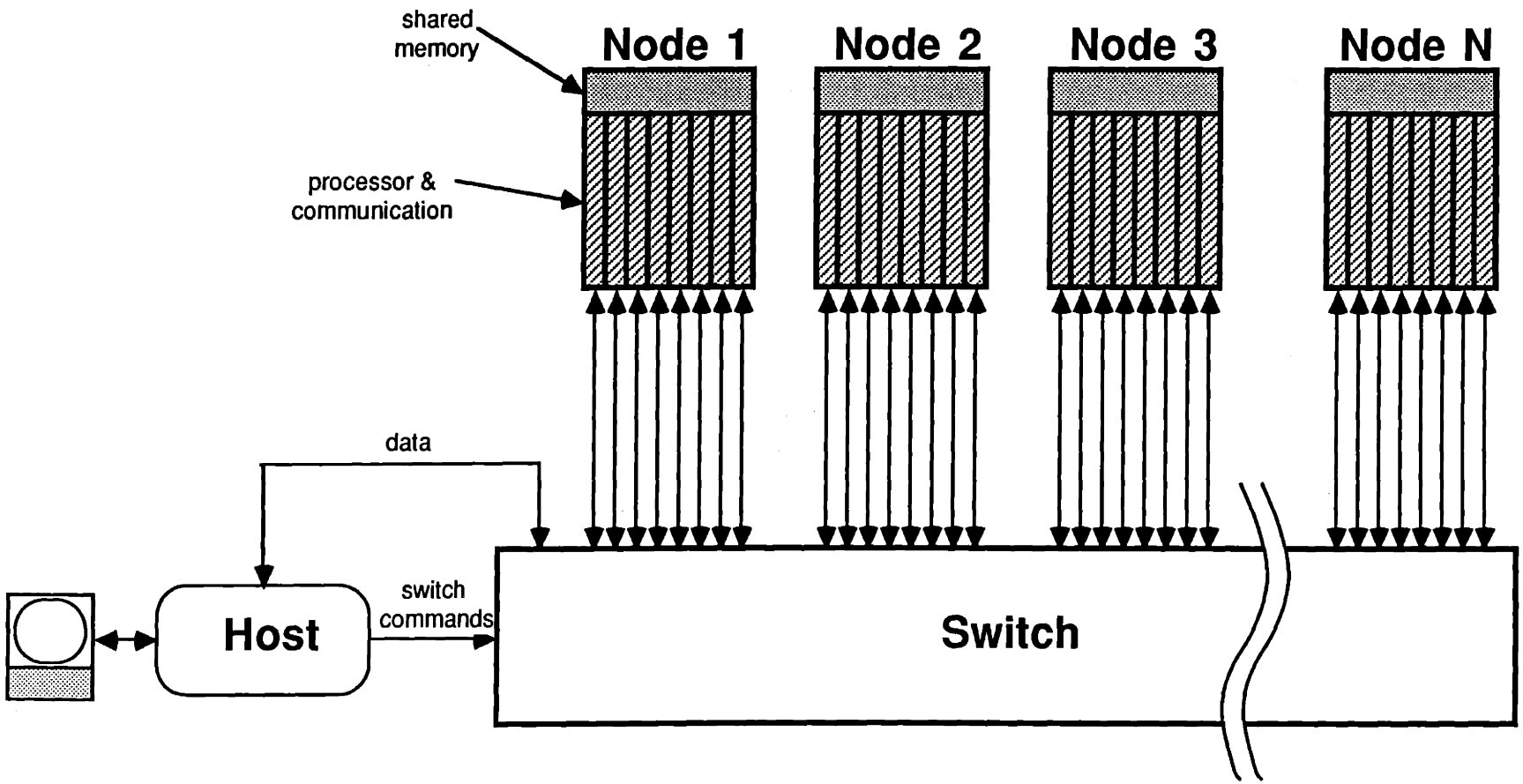


Figure 1: The (CLUMPS) System

physically possible in (CLUMPS) because all signals travel equivalent paths through the switch. Forcing the receiver's software to put in artificial delays depending on the transmitter starts to defeat the real time aspect of the simulation.

Delay is simulated in wire networks by having the transmitter wait a specified amount of time between getting the next packet ready, and actually transmitting it. Rather than exactly simulate delay in radio networks we will provide a delay mechanism in the transmitter as in the wire network case. Using this kind of delay is as if all nodes were physically the same distance from each other.

This method is good enough because we are not usually concerned about delay in packet radio networks. Delay only affects direct collisions, not indirect ones. When the delay is much less than the total packet transmission time, as it typically is, carrier sensing will eliminate almost all direct collisions. The most significant type of collision is therefore the indirect collision, which is not related to delay.

2.2.2 Hardware Strategy and Alternatives

Other simulation machine designs are possible[3], but for practical purposes there are certain features they must have if we limit ourselves to an actual network, as opposed to a large simulation program running on a minicomputer. Keeping in mind the project's goals we can draw the following conclusions about its composition.

- Interconnection

  Clearly, since we are studying the effect of different control algorithms on communication networks, we need a network that is put together in the same way as the one being studied. Bus or ring local

---

3. For example see: IEEE, Computer, vol. 15 no. 10, October 1982, Theme: "Distributed System Testbeds". Many other references and a discussion of different architectures can be found in: M. A. Tubinis, Implementation of a Data Communications Node Utilizing a Novel Multiprocessing Architecture, S.M. Thesis, MIT, 1983.

area networks are not feasible for modelling point to point connections. Similarly, point to point is not adequate for modelling radio-type connections without enormous software help. The only solution seems to be a digital switch that can be configured to provide both kinds of connections.

- Architecture

At first we considered a simple single processor architecture. Each node would have one processor with memory and many serial ports. This architecture has the advantage of low cost, and ease of programming, since commercial compilers could be used. The problem is that the processor would be overworked to handle any reasonable amount of communication traffic. Either a lot of time would be spent polling the serial ports, or in interrupt handling. Worst, the lines would not be independent, since heavy traffic on one would affect the processor's ability to cope with the others.

Another possibility was two processors: the communication processor, and the computation processor, talking via a shared memory. This has some of the same problems, i.e. the maximum communication rate per line is one eighth of what the communication processor can handle. Another disadvantage is that programming of two processors with different functions is more difficult than programming one processor with both functions. The main advantage is that computation speed is independent of low level communication overhead. The programming problem could possibly be solved by putting all of the communication processor's code in ROM, and letting the user only program the computation processor. See the last chapter for a reevaluation of this architecture in a more favorable light.

If one communication processor is good, then more must be better. The communication performance of the above scheme could be improved with more communication processors that are each responsible for fewer lines. This is a good strategy if there is more than one designer on the project, and if the tasks of the communication

subsystem can be rigidly defined. In an experimental system it would be nicer to have a lot of the low level communication tasks under user control.

The system we chose was like the above, but without a master processor, just communication processors. This has the maximum communication throughput, since each line has its own processor. All communication lines are guaranteed to be independent. It is also the most modular to design and manufacture. Programming is simplified since all processors are programmed alike. We can also experiment with true distributed processing. Processor time is not wasted, since all time not spent on the communication tasks is used for computation.

The final system then, has a number of identical processors with their own communication interface and a small amount of memory. There is also one large memory that all processors can share. This memory needs to be big to be able to handle a large number of communication buffers.

2.2.3 Software Strategy and Alternatives

The project's goals and the nature of the chosen hardware in turn give us some software goals.

- <u>High Level, Modular Language</u>

  As stated in the system goals, we want the experiments that use the system to be written in a high level language tailored for communication. The language should have common communication routines as either built-in functions or callable modules. It should be easy to tweak an experimental system to compare algorithm variants.

  The language chosen for this job is called Network Implementation Language (NIL). Besides the above qualities it has several nice features for communication software. Queues and tables are standard

data types. Many processes can be run concurrently, and they can exchange data both asynchronously (putting it on a queue) and synchronously (waiting for a reply).

The multiprocessing nature of NIL is especially useful with our multiple processor hardware. Each physical processor can work in parallel on different processes. Since most communication systems can naturally be broken down in several parallel tasks, using NIL will improve the overall processing time compared to a single thread language.

- Communication Efficiency

Communication is the major part of what the nodes will be spending their time on. Since we are only interested in packet communication, the function of the low level routines can be well defined. Usually we are not interested in the methods that nodes use to compute CRC checks, to partition incoming data into packets, etc. The implication is that these routines should be written in assembly language, and should be stored in ROM that is local to each processor. The necessary flexibility (e.g. in baud rate, error rate, and mode of operation) will be provided by the parameters that these routines can be passed during initialization.

- Code Size Efficiency

The nodes in (CLUMPS) were designed to be inexpensive, hence they use 8 bit processors, and only have about 64K of memory. Since we want to keep as much of that memory available for communication buffers as possible, and since most of the time critical code (for handling the communication hardware) is already in ROM, we can trade off processing speed against code size. The best way to do this is to have common routines or an interpreter, (which amounts to much the same thing), in ROM, and compile the user's experimental program into ROM calls. The extra step of dispatching each instruction of the program slows things down, but much more function can be packed into a given space than with assembly language.

We have chosen to take this procedure to an extreme, and to create an intermediate interpretive language specifically for NIL. All of the major NIL functions and concepts have direct counterparts in the intermediate language, so that the compilation step does not introduce any awkwardness. Each processor has an interpreter in ROM to execute the intermediate language processes.

Our solutions are not the only possible ones that could meet the software goals. Other languages besides NIL could have been chosen. We picked NIL because it had the right mix of communication and multiprocessing features, we had good access to the developers, and it was a new language that we could adjust without having to adhere to strictly established standards.

Another strong possibility was to do an interpreted language[4] like Forth. This kind of language gives the user a basic set of primitives, and incorporates user written routines into the language as new keywords that can be used in subsequent routines. It very naturally adapts itself to a split ROM/RAM environment, and generates very compact code. The main difficulties are that it is not immediately clear how to adapt to multiple processors, and execution speed can be somewhat compromised. I still am not certain that this method is a worse choice.

---

4. R. G. Loeliger, Threaded Interpretive Languages, Byte Books, 1981

Chapter 3

HARDWARE

3.1 Nodes

The computing elements (nodes) in a (CLUMPS) system are inexpensive (~$1000), microprocessor based machines with a large communication capacity (up to eight 76.8 Kbit/sec lines). Each node is a multiprocessor. See figure 2. It has three basic parts: processing units (branches), bus, and shared resources. A more detailed description of the architecture including memory maps is given in Appendix A.

3.1.1 Branches

A branch consists of a microprocessor, memory, a serial interface, a timer, and an interface to the global bus. All branches are identical. A node can have up to eight branches. The microprocessor (a 6809) uses memory-mapped I/O. Of the 64K address space about 10K is local access memory or peripherals on the processor's branch board. These are called "local" locations. The rest of the memory space is located on the shared memory board, and is referenced through the global bus.

These "global" references, while transparent to the processor, compete with global references of other branches for access (see the following section). The local resources of other branches cannot be accessed. Because of bus contention, global references are two to three times slower than local ones. This is of little consequence since the processor's machine language instructions and stack are kept locally.

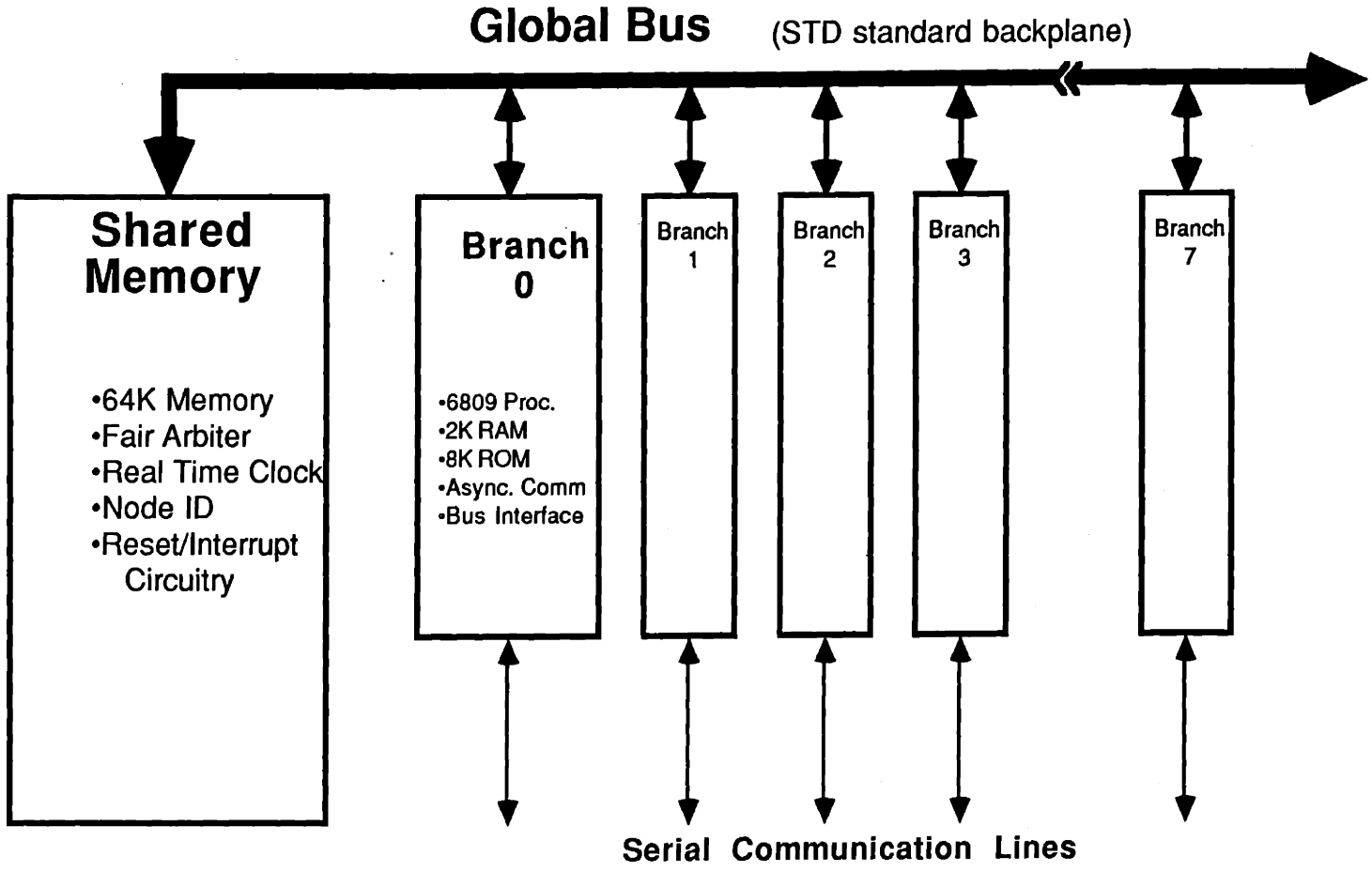


Figure 2: A (CLUMPS) Node

Besides the processor and global interface, the branch contains 8K of ROM, 2K of RAM, and a serial interface. The ROM contains all the processor's instructions: the interpreter, serial interface interrupt routines, and the operating system core. The RAM is used for the processor's stack, packet buffers, and a few variables used by the interpreter. A timer is used in conjunction with the serial interface to provide programmable baud rates, and interrupts at packet boundaries.

3.1.2 Global Bus

The global bus connects the individual branches to the shared resources. Since only one processor can connect to the shared resources at a time, the bus contains an arbiter to serialize its use. All processing units have equal priority, the arbiter ensuring that no branch gets two bus accesses while another is waiting. For an analysis of the arbitration scheme used and other alternatives see [5].

The bus interface itself was kept as simple as possible to allow flexibility in adapting future kinds of equipment. It has a 16 bit address bus, 8 bit data bus, and one read/write line. Each branch has its own pair of request and acknowledge handshake lines. When a processor accesses a global memory location, it is halted and its request goes active. When the arbiter assigns it a slot and causes its acknowledge to go active the branch removes the request, puts its address on the bus, asserts read or write, and in the case of write puts its data on the bus as well. When the shared memory has completed the cycle it signals the branch by removing acknowledge. The branch restarts the processor, removes itself from the global bus, and in the case of a read latches the data. The simplicity of the interface is a result of the asynchronous nature of bus cycles. Branch boards do not need to be synchronized with each other or with the shared memory. This increases the system reliability as well as its flexibility.

---

5. Shahrukh Merchant, The Design and Performance Analysis of an Arbiter for a Multi-Processor Shared-Memory System, MIT MS thesis, Sept. 1984

3.1.3 Shared Memory

The shared resources (sometimes called global memory) consist of a large memory and a number of memory mapped special functions. The memory is used to hold the intermediate language routines, communication buffers, statistical results and other data used by the experiment.

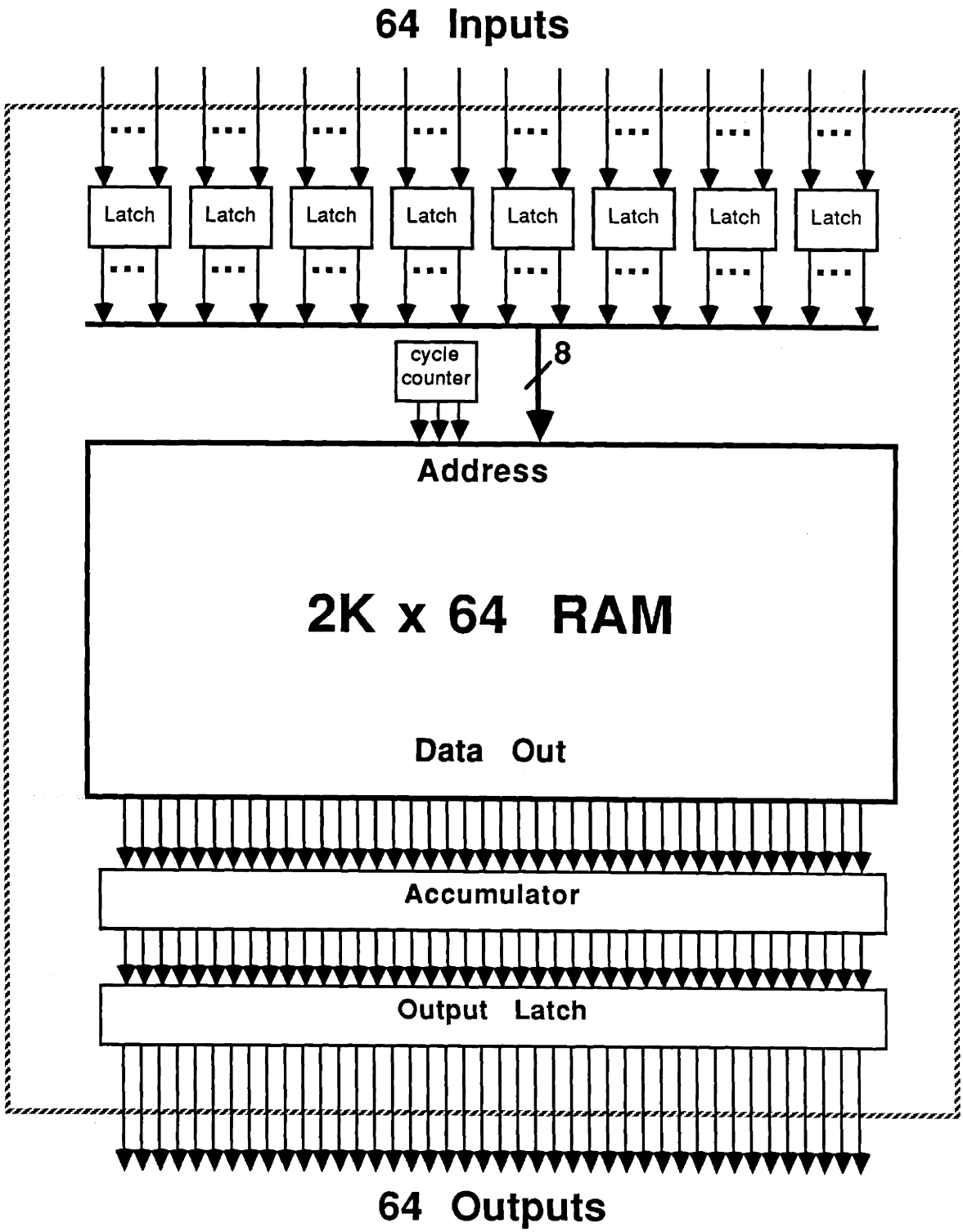
The other major resource on the global memory is a 48 bit counter clocked every 400ns, which is used as a real time clock. This allows the branches to have a consistent timing reference. Also among the shared resources are locations for reading the node and branch id numbers, and locations that cause system interrupt or reset. These are used to recover from failures, or to reinitialize the system for new experiments.

3.2 Switch

The ability to simulate arbitrary networks comes from the specially designed digital switch that connects the computer nodes together. The switch can make radio-type connections, i.e. combining many nodes' outputs to form an individual node's input, as described in the previous section on modelling. Wire networks are a subset of radio networks where a particular channel exclusively connects two nodes. The switch is designed to handle up to 64 bidirectional communications lines, each at a rate of up to 76.8 Kbits/sec.

The switch is unusual in that it does its space-division steps via table lookup, not multiplexors. The entire set of 64 inputs is switched to the 64 outputs in eight time division cycles. See the switch block diagram (figure 3). Each cycle takes eight inputs, and creates 64 outputs, by table lookup in a 256x64 bit memory. The results of the eight cycles are combined in a 64 bit accumulator. This performs a logical OR of the 64 lines. After eight cycles the results are stored in a 64 bit latch, and the accumulator cleared. Since each cycle uses its own table, the

Figure 3:
Digital Switch Data Flow



total memory size is 2K x 64 or 16K bytes. There are also some buffers so that a microprocessor can read and write the memory.

The advantage of this scheme is that it is very fast (eight times faster than TDM), it can combine lines in radio fashion, and it uses very few parts (less than a pure TDM for less than 512 lines). Increasing the number of lines that the switch can handle only adds an extra slice for those lines, and increases the memory size for the others. An increase in memory size can often be accomplished at a very low cost. The disadvantage is that to change a connection one must alter 128 bytes of memory (on the average). This is not a problem when simulating wire networks, but it may be in very large radio networks with mobile communications. Our switch requires about 2 milliseconds to make a connection. The fact that a connection is not made instantaneously is probably a good model of actual radio networks.

The above discussion uses figures from the design that we implemented[6]. They represent good trade-offs in speed versus cost for current technology. Other applications of the basic idea (switching using table lookup) are possible.

3.3 Host

The host is an external computer used for creating and compiling experimental programs, the operator's interface, and permanent storage of programs and data. All of the programs that run on it are written in C, as portably as possible, so that the actual choice of machine is not critical. In our case we are using IBM PCs. The host connects to a node's serial port using a standard RS232 interface.

---

6. Stephen LeBlanc, 6.100 Lab Report (not published), June 1984

Chapter 4

SOFTWARE

The (CLUMPS) network was not designed for software development, but to be a simulation peripheral. (CLUMPS) augments the host for the purpose of communication network simulation; it is not a complete computer system. The user's programs are developed and compiled on the host.

The language chosen for writing the simulation programs is called Network Implementation Language (NIL), and was developed by IBM[7]. This is a high-level language, with similarities to Pascal and Ada, but with special features designed for communication work. This language has data structures commonly used in communication systems such as queues, messages, and tables. It has multitasking and well defined interfaces, to facilitate layered communication system design. It does not burden the user with implementation details such as the exact representation of data in memory.

A NIL program is semantically well defined, as all operations have specific outcomes regardless of the data involved. As a simple example, in many languages addition is actually performed modulo the processor's word size. This means that a program that adds 30,000 to 30,000 may come up with 60,000 on a 32 bit machine, and -5,536 on a 16 bit machine. The same program in NIL will encounter a DEPLETION exception on the 16 bit machine, indicating that resources were not available to perform the desired action. Exceptions can be handled by the user program, or reported to the operating system.

---

7. Draft NIL Reference Manual, RC 9732, IBM T. J. Watson Research Center, Yorktown Heights, NY, Dec. 1982

There are two shortcomings of having such rigid semantics at the lowest level. First, implementations are not likely to be very efficient because a test for illegal conditions is necessary after every arithmetic operation. This is not a big problem in our case, since by using an interpreter, the extra overhead is comparatively small. The second problem is that semantics of real numbers and their operations are difficult to define, and in fact NIL does not currently support real numbers.

Our implementation is designed with a combined compiler/interpreter and an intermediate language similar to Pascal's P-code[8]. The compiler in the host creates intermediate language object code, which is loaded into the nodes' shared memory for execution. Each branch contains an intermediate language interpreter in ROM. NIL is a multiprocessing language. A list of active processes is kept in shared memory. When a branch's interpreter is free it finds the next process on the active list and executes it. Processes are not fixed to a particular processor.

In addition to the user's processes, a running system also has an operating system. The operating system is a collection of low level routines that allow the user's program to execute multiple tasks, exchange packets with other nodes, and report results to and receive data from the host. The OS also performs services such as system initialization.

Some of the operating system is in the processors' ROM, and some is loaded when the system is started. The ROM based part operates concurrently with the interpreter when dealing with the communication port, and separately for larger tasks, such as allocating buffers. Since it is written in assembly language, it is not meant to be changed by the user, and care was taken to ensure that its communication facilities are general enough to satisfy a broad range of simulation needs.

---

8. Philip Nelson, "A Comparison of PASCAL Intermediate Languages", SIGPLAN Notices, Vol. 14, No. 8, Aug. 1979, pg. 208

4.1 Network Implementation Language

4.1.1 Official Language Specification

Most of the information presented here on the "official" NIL specification can be found in more detail in the NIL Reference Manual[7]. It is only repeated here to familiarize the reader with the features of NIL, and to motivate the changes made to the language for the (CLUMPS) project. A number of the goals and features of NIL (such as portability and security) are not relevant to this project (which is a special-purpose, single-user system) and will not be discussed.

The three features of NIL of most interest to us are:

1. Nil programs are close to the functional design level, and do not include specifics of the memory structure of data, which is left up to the compiler. This means that users can concentrate on the functionality of their algorithms, not on the details.

2. Modules are separately compilable and interact ONLY through explicitly defined interfaces. Common problems of misusing programs written by others are eliminated. Debugging is easier since modules cannot inadvertently affect other ones. There are no global variables, so that libraries of independent routines can be built.

3. NIL uses a process model with both synchronous and asynchronous actions. Communication systems normally are performing several functions at once, and with up to eight processors, the (CLUMPS) hardware is set up for multiple processes. The NIL model is simple and complete.

On the surface, NIL is similar to Pascal[9], with its type and variable declarations and block structure. However instead of variables representing a certain memory structure, variables are abstract objects which do not specify any particular implementation. For example, the concept of a NIL table, which is an aggregate of arbitrary information, is a generalization of Pascal's array, which is only a fixed, rectangular block of memory. Arrays are just one possible implementation of tables, others could be linked lists or hash tables. It is also possible for different implementations to coexist, with a particular one chosen by the compiler based on the specific situation.

4.1.1.1 Objects

Before describing the data types that NIL provides it may be helpful to explain the syntax of type and variable declarations. This should eliminate any confusion over the examples that will follow.

As in Pascal, variables in NIL can either be given built-in types, or have user defined types. Types are defined with the IS keyword, and variables are defined with a colon (:). Types are defined first, then variables are declared after the DECLARE keyword. For example the following declares the integer i: (Throughout the text we will highlight NIL keywords by capitalizing them; this is not required.)

```
DECLARE
        i: INTEGER;
```

The same thing could have been accomplished with a user defined type:

```
        itype IS INTEGER;
DECLARE
        i: itype;
```

---

9. K. Jensen and N. Wirth, Pascal User Manual and Report, Springer-Verlag, 1976

In NIL the built-in types are called type families. This emphasizes the fact that some of them are merely ways of building structures, and are not specific types until they are used. For example, the MESSAGE type family is used to group variables of different types, much like the record type in Pascal. For example, we can group an integer and a character string as follows:

```
m1 IS MESSAGE
   (i: INTEGER;
    s: CHAR STRING);
```

In this example m1 is a particular type, but MESSAGE is its type family.

NIL defines the following object type families: integer, boolean, enumeration, string, table, row, tableset, variant, message, call message, send interface, call interface, catalog, and component. Integers and booleans are standard, except that the maximum size of integers is not defined. If ever a program tries to exceed the maximum size, an exception is raised since NIL does NOT perform modulo arithmetic. Exceptions are recoverable. Enumeration variables are defined to take on one of a fixed list of values. The type color = (red, blue or green) is an example. Another is the Pascal standard type char which has 256 possible values. Strings are lists of either enumerations or booleans.

Send and call interfaces are the objects used to transfer data from one module to another. Call interfaces operate like subroutine calls in most other languages. One module sends a call message to another, waits while the information is processed, and then continues after the results have been returned. This is called synchronous communication, because the two program modules work on the data in a specific sequence, rather than in parallel. The call interface consists of two variables, one owned by the calling module, and the other owned by the called module. The calling module has a callport, which is really just a pointer connecting it to the other module. The called module has an acceptport which is a queue in which calls can accumulate. Many callports can be connected to a single acceptport (just as many routines can call a common subroutine in most languages).

Send interfaces are asynchronous. One module sends information to another, does not wait, and does not receive a reply. The interface consists of a sendport connected to a receiveport. As in the synchronous case many sendports can be connected to the same receiveport. The receiveport queues up the messages sent to it, until the module that owns it uses them.

When discussing the NIL communication mechanism we often use the term primary to refer to both acceptports and receiveports, and the term secondary to refer to callports and sendports. This is because acceptports and receiveports are the complicated part; they are queues that can receive messages from multiple sources, and hold them for their owner. A secondary is just a pointer to one of these queues.

A catalog is used to keep a list of primaries together with their names (character strings). It is really a specialized form of table. When the sending half of an interface wants to connect itself to a receiver, it gets the reference from a catalog in which the receiver has already published its name. The connection is made at run time, so that information can be exchanged between separate modules without linking.

Messages are similar to records in Pascal, and variants are like variant records. These are simply collections of objects with a fixed structure. For example, a message type consisting of a group of two boolean strings and an integer would be declared as:

```
group IS MESSAGE
   (s1  : BOOLEAN STRING;
    s2  : BOOLEAN STRING;
    i   : INTEGER);
```

Messages can be used to group data to be sent to other processes over a send or call interface. Messages used for this purpose are declared as messages or call messages respectively. They are given a different type family to emphasize the subroutine call and return use of the call message.

Components are very simple objects which represent a running module. They allow processes that generate subprocesses to maintain control over them. When a new module is spawned, its parent receives an initialized component. Only two other operations are possible on a component: canceling it (which involves stopping any processes in it and freeing up their resources), and checking to see if all processes in the component have finished. Processes can interact with their subprocesses by passing them parameters when they are initialized.

There are only two ways to accumulate data into an aggregate, strings and tables[10]. A table is a collection of records, each of which has the same structure. A group of related tables is called a tableset. Tablesets, tables and rows are based on a relational database model [11]. The specification for a table lists the types of all of the fields of its records, and a few extra attributes that tell the compiler how the table should be organized. For more information on tables and our original implementation of them see [12].

Information in tables are accessed by objects called ROWs. In essence a ROW variable names one of the table's records. Rows are implemented as pointers, so they can refer to different records at different times. Rows are the handles used to manipulate the data in tables. It is important not to confuse a ROW variable with the actual data record it is pointing to. We call the data itself a tuple. While a table can have an arbitrary number of tuples, each ROW variable associated with that table is explicitly declared.

---

10. Actually primaries (queues) can also be thought of as data aggregates, but we choose not to because of their special use in communication.

11. Jeffrey D. Ullman, Principles of Database Systems, Computer Science Press, 1980

12. George Nikolaou, Table Implementation for the Network Implementation Language (NIL) Compiler, MIT BS thesis, Feb. 1984

4.1.1.2 Operative Statements

NIL has the standard operations on booleans, integers, enumerations and strings. Expressions can also be created using built in functions such as LENGTH, ORD, SUCC, etc. NIL supports two kinds of assignment statement: copy and move. Copy assignment gives the variable on the left hand side of the statement a value, and doesn't affect any other variables. Move is used to transfer data to one variable, and remove it from the original variable.

Moving data can be much faster than copying it. For example, string variables are usually implemented as pointers to a memory block. Move would just manipulate pointers, while copy would have to allocate another block and copy all of the data. Move is supported for all type families, while copy is only valid for the boolean, enumeration, integer, string, callport, sendport, catalog and row type families.

The DISCARD operation can also be used with all type families. It simply throws away a variable's value, and returns its resources to the system. Its opposite is ALLOCATE which is used to create resources for the structured types (messages, variants, and rows). Simple types do not need to be allocated, i.e. their resources are static.

The objects related to communication, messages, ports, and catalogs are operated on by the ACCEPT, CALL, CONNECT, FORWARD, PUBLISH, RECEIVE, RETURN, SEND and UNPUBLISH statements. PUBLISH puts a reference to a port into a catalog under a character string name. UNPUBLISH removes a reference from a catalog. CONNECT uses a catalog and a string to initialize a callport or sendport.

Asynchronous communication is accomplished via SEND and RECEIVE. After a sendport is CONNECTed to a receiveport, SEND is used to put a message onto the receiveport's queue. RECEIVE is used to take it off the queue.

The corresponding instructions for synchronous communication are CALL and ACCEPT. A CALL puts a call message in an acceptport's queue, and

causes the calling process to wait. The called routine later does an ACCEPT to get the call message, processes it, and performs a RETURN to send it back to the caller. The caller then continues execution. The called routine can use FORWARD to pass the call message on to another process' acceptport. When the message is later RETURNed it goes directly back to the caller. This allows the construction of processes that route and schedule calls, or ones that pipeline the processing so that calls are executed in multiple steps.

A very useful operation in handling communication problems is changing data from one format to another. For example a machine that uses 8-bit numbers may send information to one that more naturally uses 32 bits. NIL lets the user specify an exact definition for the number of bits used by each field in a message. This specification is just used for external communication to another computer. The REMAP command is used to translate a message between internal and external form. It is very versatile and can also translate between two different message formats. Almost all of the work of decomposing a message containing layered protocols can be quickly handled by REMAP.

CREATE is used to start up a new module given the module's name (as a character string), and some initializing parameters. The object code for the named module is retrieved from the operating system, and started. When the module is fully initialized, the parent routine resumes execution, and its parameters are returned. The parent also gets a component variable that controls the new module. If the component is DISCARDed then all processes in the module will stop, and all of their resources will be freed. Note that the compiled modules are kept by the operating system in one global database.

Several operations deal with ROW variables. Tuples can be DETACHed from the table and DISCARDed. New tuples can be ALLOCATEd and INSERTed into the table. A ROW can be disassociated from a tuple with the LOSE operation.

The FIND operation is used to associate a ROW with an individual tuple based on that tuple's key or position in the table. When a FIND is

performed, an access mode is specified to determine the level of access to the tuple. Three access modes are possible: refer, read, and update. Update is for read/write access, read for read only access, and refer allows no access to the data, but prevents it from being deleted. The reason for all of this is that several ROW variables can point to the same tuple, and to ensure the data's integrity it is necessary to arbitrate between them. We want to guard against having two ROWs point to the same tuple, throwing away one of them, and then using the other to look at data that is no longer there.

4.1.1.3 Control Statements

Control statements can alter the flow of program execution. We have already discussed those statements that deal with other processes and procedures (CALL, RETURN, CREATE). NIL is block structured like Pascal and other modern languages. Like Pascal it has IF THEN ELSE, REPEAT (for looping), and SELECT (like CASE) statements. Many of these standard control statements have been extended to take advantage of NIL's rich data types. REPEAT has a version that can loop over selected records in a TABLE. SELECT can cause a process to wait until one of a number of external events take place.

NIL has an extensive exception handling mechanism. Every possible exceptional condition, ranging from an addition overflow to a data type mismatch in a procedure call, can be caught and handled by the user's program. The

```
DO
   block
ON (exception)
   handler
END DO
```

structure is used to define a block of code that will share a common exception handler. Exception handlers can be nested. When an exceptional condition occurs control is passed to the innermost ON statement that

handles that particular condition. If an exception occurs in an exception handler itself control is similarly passed outward to the next appropriate handler. If no handler exists for that type of exception the process is terminated, and the exception is passed on to its parent.

This way of processing exceptions together with the extensive compile time typestate checking of a NIL program goes a long way toward ensuring that a program is semantically correct. If the program flows normally it means that every statement had the exact result that the programmer had in mind. No bad pointers can be used, no integer counters can overflow, and so on.

The one especially tricky exceptional case occurs when a process is cancelled by its parent. NIL guarantees that the process will halt in some finite time and all of its resources will be returned to the system. A potential problem occurs when the cancelled process is currently calling another process that will not be cancelled. Some of the calling process's resources (namely the CALL MESSAGE) are no longer under its control. The caller cannot be cancelled until the call message is returned. The way NIL handles this problem is that the FORCED exception is raised in the called process, and is not cancelled until the offending call message is returned. While the FORCED exception is active the process is not allowed to loop, wait, or call any other process.

4.1.2 Language Modifications

(CLUMPS) special situation as a self-contained single user microprocessor based system warranted many changes and simplifications to NIL. The biggest philosophical change was to relax the rigorous checking done to insure that programs interface correctly, do not invade one another's data, and do not use uninitialized variables. These checks add considerably to the complexity of the compiler, and in our system only serve to save the user from himself, not from others. This type of checking can also be done by a separate program (such as Unix's lint program which checks C programs), and thus such checking is not necessary within the compiler. Other changes reflect a reduced scale, more

appropriate for a small system or are simply improvements over the original language.

The VARIANT data type was eliminated. This type is used primarily to distinguish between the presence and absence of data, as in the recursive type:

```
list_type IS VARIANT                        { type definition }
        (case (empty) : ()                  { structure }
         case (full)  : (i : integer,
                         next : list_type));

DECLARE  list : list_type;                  { variable definition }
```

This is how NIL could create linked lists and trees even though it does not have explicit pointer types. In our version this type of structure could be made from messages because of the lazy initialization scheme (see section on implementation). The equivalent definition in (CLUMPS) NIL is:

```
list_type IS MESSAGE (i : integer, next : list_type);

DECLARE  list : list_type;
```

A built-in function, EMPTY, is provided to test whether an object has been initialized. In this case one would test for the existence of more data with the expression EMPTY(list.next). Lists are still not as useful as tables for storing groups of data, because there are no pointers in the language. To move along a list one would have to constantly break it up with MOVE statements. Using tables and rows is preferable.

We made several simplifications in the area of tables. First, ordered tables were eliminated. This limits the ability to immediately find a row given its position (e.g. finding row number 7) and to compare rows to see which one is first. Either operation can be accomplished by using a key that is the position number. We still can step through a table by using FIRST, LAST, BEFORE and AFTER constructs.

The biggest simplification was the elimination of ROW access modes. These modes were used in NIL to ensure that multiple ROWs didn't point to the same tuple in a harmful way, and the compiler could always keep track of what was going on. Since our interpreter makes sure that all ROW operations are valid at the time they are executed, the compiler doesn't have to worry. Access modes were one of the most complicated parts of NIL, and were only there to catch rare semantic errors.

Another significant simplification was not to allow tables and rows to be top level objects; they must be contained in a tableset. NIL had a lot of problems resulting from tables and rows being independent objects. Situations could occur where tables were passed to other routines while they had open rows, or tables could be moved without closing associated rows, etc. We eliminated all of these problems by making tablesets contain both tables and rows, and any transfers affect the entire tableset. Tables cannot be individually discarded; one can only get rid of the entire tableset. This prevents linked tables from having references to information that has been thrown away.

NIL was designed to allow process and function resources to be grouped into catalog objects. Catalogs allow separately compiled programs to be linked together at run time. They also let users hide their resources from one another (by keeping them in different catalogs). Since (CLUMPS) is a single user system, we simplified this to having just one global catalog that is implicitly referenced during PUBLISH, UNPUBLISH, and CONNECT operations. Programs can still be separately compiled, but there is only one place that they can be listed.

The last big change was eliminating procedures and functions, and limiting modules to contain only a single process. This means that the only kind of program module is a process. This is not as restrictive as it sounds at first. Processes can still be called using callports and acceptports just as one would call a procedure or a function. The only cases where this is not acceptable is when recursion is involved, or when several processes are calling a subroutine at the same time. If they are calling a process their calls would be handled sequentially, while if they were calling a procedure the calls could be handled in parallel with more

efficiency. As explained below, the structure we have chosen can also handle these cases.

In NIL, processes run in two phases. The first is called the create phase, which is used to initialize the process with parameters from its parent. The second is called the main phase, during which it is a fully functioning process. The sequence of actions by the operating system, parent and child processes in starting up a new process is as follows:

Parent	Executes CREATE, passing parameters in call message
Opsys	Gets object code from host if necessary, starts child process
Child	Runs create phase with parent's parameters (like a call)
Child	Returns parameters (in call message), continues with main phase
Parent	Gets back parameters (possibly modified by child), gets component variable pointing to child, continues with next instruction

Notice that the create phase of a process acts very much like a subroutine. Parameters are passed to it, they can be modified, and they are passed back. A process without a main phase is just like a procedure, while one without a create phase is a fully independent process. When a process is not sufficient, and a subroutine is necessary (or desired for efficiency), we can still use a process that does not have a main phase, and access it with CREATE instead of CALL.

We also introduced a few minor changes and improvements to the language. Rather than having a sophisticated compiler that decides when different number sizes are necessary, we took a more traditional approach and added the LONG data type for 32 bit numbers (INTEGER is for 16 bit numbers). We also added the long() and integer() built in functions to allow the programmer to decide the precision necessary inside of expressions. The compiler will convert between sizes when necessary, however.

The enumeration data type has been scrapped in favor of the CHARACTER type. Like Pascal, conversion between integer and character is possible

with the ORD() and CHR() built in functions. Strings of integers and longs were added to give more flexibility in the kinds of aggregates possible without incurring the overhead of tables. The new ZEROSTRING instruction allows the user to form strings of zeros of a given length in one step. The substring semantics was changed from a low and high index to a low index and a length.

The RECEIVE instruction was changed so that it gives an exception instead of waiting if the given queue is empty. Waiting can be done separately by the WAIT instruction.

REMAP was changed to work with a character string and another object, rather than two messages.

The SLEEP, COMCALL and COMSEND instructions were included to interface with the operating system. SLEEP puts the current process on the end of the process queue (essentially gives other processes a chance to run). COMCALL and COMSEND are used to send communication messages to serial ports.

The FORCED and CANCEL exceptions were eliminated. Now when a process is cancelled, it just stops immediately. The OVERFLOW exception was added for arithmetic overflow errors. DEPLETION is now mostly used for memory allocation failures. The new PARAMETER exception is used for situations that are now caught by the interpreter, that would be caught by the compiler in a full implementation.

A number of new built in variables have been included. RANDOM and RANDOML return integer size and long random numbers respectively. TIME returns the current time in 104.2 microsecond units (it is a long). The time can also be set by an assignment statement. NODEID is the single character node address. BRANCHES returns the number of communication lines available to the user program, and PROCESSORS returns the total number of processors in the node.

4.2 NIL Implementation

Our implementation of NIL splits the job of compilation into three parts: the compiler, the assembler, and the interpreter (see Software Strategy section above). The common thread of these three pieces is the Intermediate Language. This language is used to represent the NIL program in a form convenient for the interpreter. The concept is the same as P-code to Pascal, but the language is very specific to NIL, and has many more opcodes. The compiler takes the NIL source, and converts it to intermediate language source. This is then assembled into intermediate language object code, which can be interpreted. Since the intermediate language source can have symbolic labels we use a one pass compiler, and a very simple two pass assembler.

This sequence of events is distributed as follows: the compiler and assembler run on the host to generate the intermediate language object code from NIL source. Object code is loaded into the shared memory of all the (CLUMPS) nodes by the operating system. Interpreters, located in ROM on all of the branches, are assigned processes. They execute these processes by reading the object code from the shared memory one instruction at a time. A branch's local memory is used for communication buffers, stack space, and internal variables of the interpreter. Since all of a process's code and data are kept in shared memory, processes can be moved from branch to branch with a minimum of overhead.

4.2.1 Intermediate Language (IL)

This language was designed with one goal in mind. It had to have a compact representation in object code. (CLUMPS) nodes have a limited amount of shared memory (64K), and small programs are very important so that there is space left for communication buffers. Small amounts of code also reduce the amount of traffic on the shared memory bus, since the processors will not be accessing code as often.

This goal was solved by having a language with a one for one statement correspondence with NIL in most cases. The major exception is expressions. IL statements are quite compact, consisting of a one byte opcode followed by parameter descriptions. Expressions are calculated using a stack maintained by the interpreter, which eliminates the need for doing register allocation, and simplifies the use of temporary variables. Expression evaluation is very similar to FORTH[13] and consists of one IL statement for each operation, and one to load each operand.

The implementation was complicated by a decision not to store the task's stack in shared memory during a subroutine call or a task switch. The other alternatives were to operate the stack out of shared memory, or to allocate a spot in shared memory for the stack at the time of a task switch. The first is very slow, since the stack is often used, and the second involves a lot of allocating and deallocating of small amounts of memory (which takes time) and is not guaranteed to succeed. The result is that the compiler has to be careful to execute function calls before other parts of an expression, and keep the results in temporary variables until the expression is calculated. (This is discussed further in the compiler section.)

4.2.1.1 Object Code

A process's object code is a static block created by the assembler. It is relocatable (i.e. all internal pointers are relative to the start of the block) so that the same code can be copied to several nodes and run directly. There are two parts to the object code: descriptors and code. Descriptors tell the interpreter about the structure of the data used by the process. They are created from the NIL program's data declarations. The code includes standard statements and exception handlers. The structure of an object code block is diagrammed in figure 4 as part of the component data type.

---

13. Byte, vol. 5, no. 8, Aug. 1980, Theme: FORTH

Figure 4:
Memory Structure of the Data Types

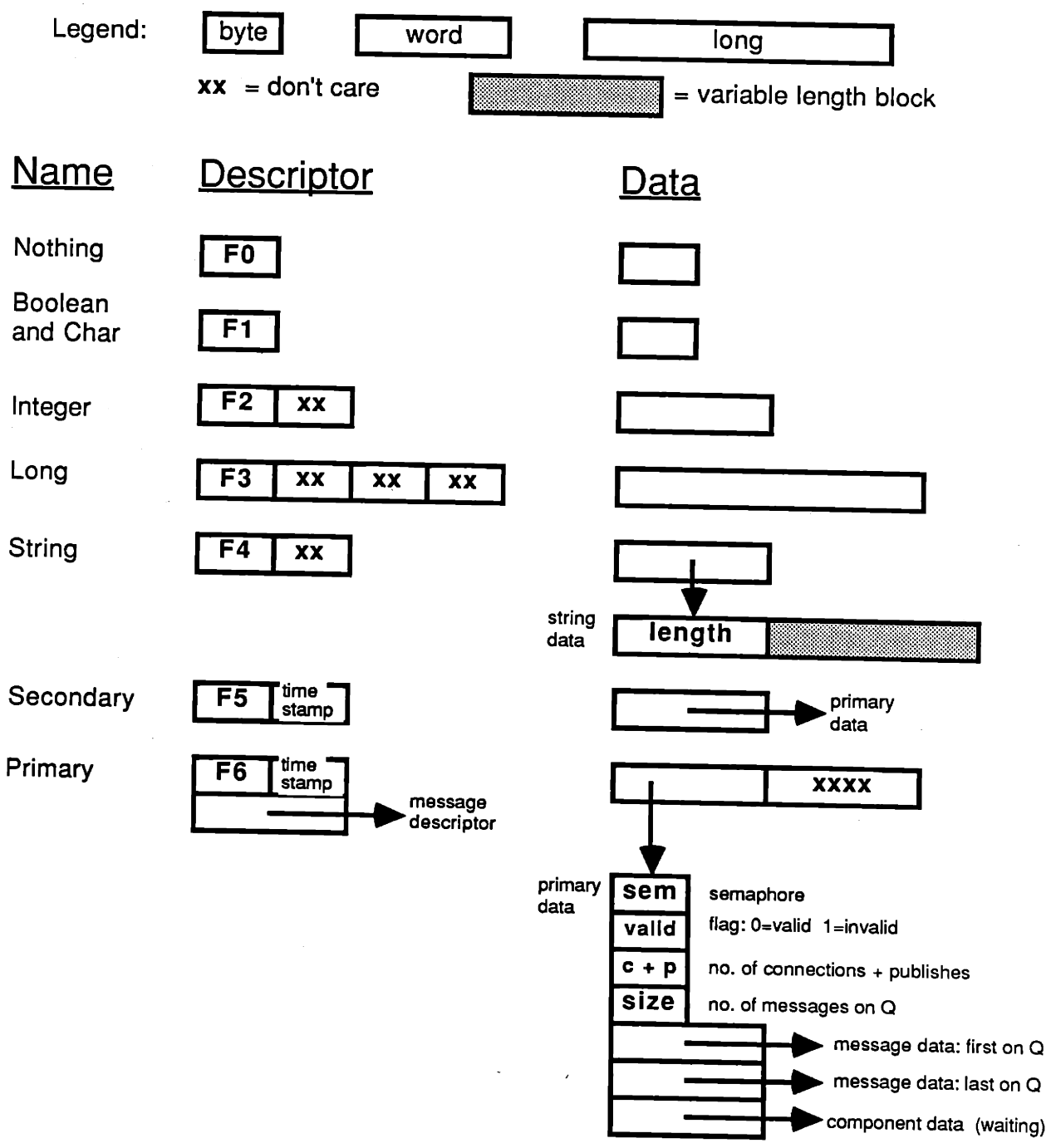


Figure 4 continued

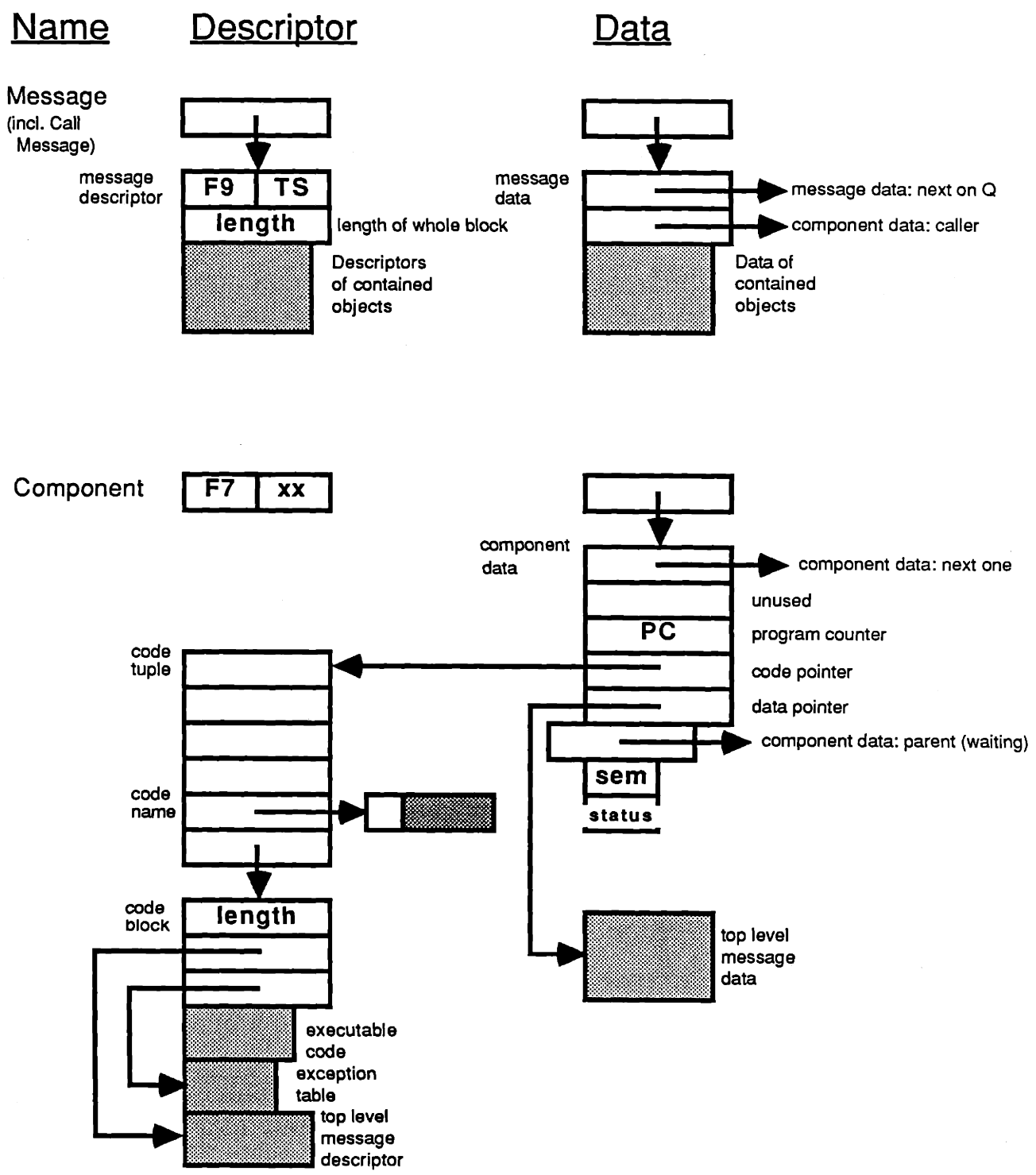
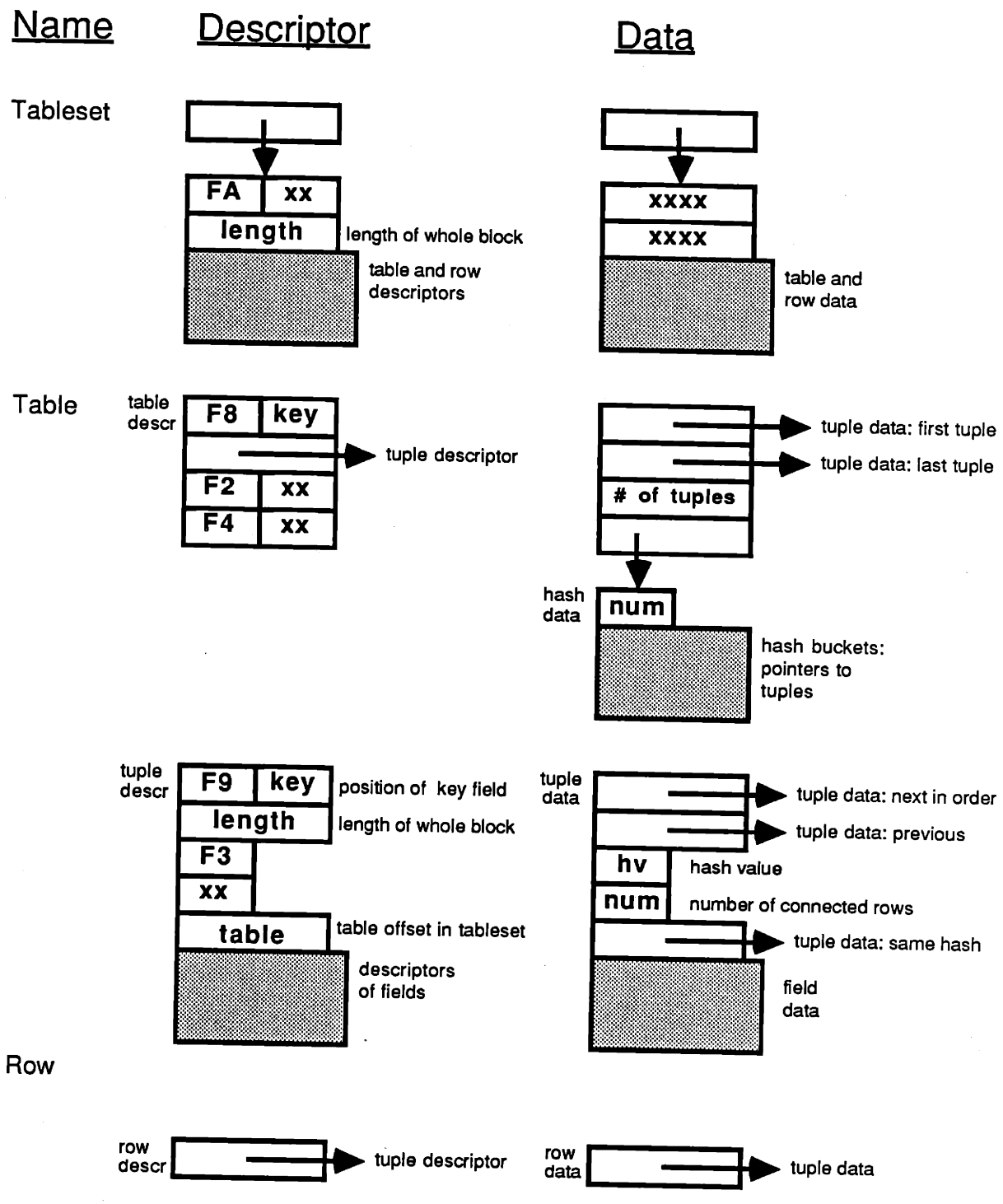


Figure 4 continued



4.2.1.2 Descriptors

NIL's object oriented structure implies that all data must be dynamically allocated. In our implementation the chores of allocating and deallocating data are performed by the interpreter, not the programmer. This insures a level of data integrity that would normally be missing or provided by garbage collection or extra compiler checking. To perform this service the interpreter must know the data's structure, hence the descriptors[14].

A side benefit of having the interpreter handle memory allocation is that operations can be generic; there doesn't need to be a version for every data type. For example, there is a single store operation, not one each for one, two and four byte variables.

Descriptors for the simple data types (boolean, character, integer, long, string, secondary, primary, component and table) are coded by a byte starting with an "F" (in hexadecimal). The compound data types (message, row and tableset) have descriptors that are relative pointers to more complex descriptor blocks. This is done when the size of the descriptor is variable because it has contained fields. No descriptor is allowed to be greater than 255 bytes long. A list of the data types and their descriptors is given in figure 4.

4.2.1.3 Data and Addressing

The descriptors are static, like code. Data, on the other hand, is allocated dynamically. The descriptors have been designed such that their corresponding data takes up the same amount of space. For example, the

---

14. The idea for using descriptors in a NIL interpreter was originally proposed by R.M. van de Weghe (and his professor A.J. van de Goor) in DUNIL, Delft University Network Implementation Language, 051560-28(1982)24, Afdeling def Elektrotechniek, Technische Hogeschool Delft, The Netherlands. Their version was too unwieldy for our use, however.

descriptor for a character is just one byte long, while for an integer it is two bytes long. When a message (or other dynamic data type) is allocated, its data block will be the same length as its descriptor. The data for the simple types (char, integer and secondary) is located directly in the data block. For other types the data block contains pointers to additional blocks that are allocated as needed. This data structure is illustrated above along with the descriptors.

Since the code is relocatable, all pointers within it are relative to the start of the code block. The data, however, uses absolute pointers to get to its blocks. All data in a node is allocated using a common routine that partitions shared memory. This routine uses a semaphore in shared memory to coordinate requests from more than one processor.

The reason that descriptor and data blocks are kept the same size at the top level is so that a piece of data can be accessed with a single offset. When applied to the current descriptor block this offset gets the variable's descriptor, and when applied to the corresponding data block it locates the data. This means that a variable can be completely described in as little as one byte, thus meeting our goal for compact code.

While top level variables can be addressed with a single offset, lower level variables are addressed with a list of offsets. One offset is used for each level of structure that has to be traversed to reach the variable. The value used for an offset is the sum of the sizes of the descriptors of the previous variables at that level, plus one. This means that the offset does not determine the number of previous variables at a level, but the total size (in bytes) of their descriptors. The first variable has an offset of one, the second one plus the size of the first, the third one plus the size of the first two, etc.

If the variable found by the first offset is a scalar, then it is the one being addressed. If, on the other hand, it is structured, then its data part has a pointer to another level, and another offset is used. When it is necessary to address a structure instead of a field in a structure an offset of zero is used. This tells the interpreter not to go any deeper. For example, consider the following declaration:

```
DECLARE
     i: INTEGER;              {two bytes, offset 1}
     c: CHAR;                 {one byte,  offset 3}
     m: MESSAGE               {two bytes, offset 4 0}
        (
        b: BOOLEAN;           {one byte,  offset 4 1}
        s: CHAR STRING;       {two bytes, offset 4 2}
        );
     s: INTEGER STRING;       {two bytes, offset 6}
```

Remember that zero is a special offset, so that i and b start at offset 1.

Each offset is described with a single byte. This means that each level of structures can have up to 255 bytes of descriptors. Since descriptors are about two bytes long on the average, we can have about 128 variables per level (in particular on the TOP level). This should be plenty for a program.

4.2.1.4 Operations

The Intermediate Language's opcodes can be broken into three major categories: those involving the stack (used to calculate expressions), those coming directly from NIL commands, and those special to the CLUMPS operating system. All opcodes are one byte long.

The stack holds 2 byte words or pointers. Some operations consider the top two stack elements to be a single 4 byte value. The stack is local to a processor, making it more efficient for calculations than allocated variables, which are stored in shared memory. All of these operations deal with the top few items on the stack.

Except for some complex control structures such as FOR loops or SELECT statements, NIL opcodes are in one to one correspondence with NIL statements.

All operations are listed in Appendix B.

4.2.2 Compiler

To make the compiler as simple as possible we put some restrictions (mostly syntactic) on NIL source code. The major ones are: regular syntax[15], requiring all variables to be declared at the beginning of a process, removing variant data types, and a simplified select control structure.

To make it more flexible, the compiler was generated with a compiler generating tool that makes a compiler from the NIL syntax (given in Appendix B) and a library of code generation routines.

The first things in a program are the declarations. These are scanned by the compiler, and a symbol table is generated. The compiler's error recovery mechanism is not very sophisticated, and a syntax error in the middle of a declaration usually causes the compiler to lose its place in the syntax. The symbol table is the compiler's equivalent of the descriptors used by the interpreter, but it also contains the names of variables and types. Type declarations come before variable declarations. When the declarations are finished forward type references are resolved. If there are any unresolved types a search is made of the global type library. Error messages are generated for any types that are still undefined.

The other two sections of the program, the CREATE PHASE, and the MAIN PHASE, contain the process' code and are treated identically. A return of the creation call message is automatically generated between the two phases.

Any place where a variable's value is needed an expression can be used. Expressions can contain variables, function calls and built in operators. The interpreter maintains an expression stack to hold the

---

15. K. L. Clark and D. F. Cowell, Programs, Machines, and Computation, McGraw-Hill, 1976, pg. 147

intermediate results of operations. All of these calculations deal with booleans, characters, integers, longs, or strings of these types. Operations are performed in operator precedence order. Like Pascal, all expressions are strongly typed.

The compiler's expression handling mechanism is a bit more complicated than most due to the decision not to save a process' expression stack during function calls. This decision was made to streamline the process swapping mechanism. It also obviates copying data from local memory (where the expression stack lives) to shared memory (where a process lives while it is waiting). The result is that when any function call takes place the expression stack must have been clear first. This may seem impossible given the fact that function calls may appear anywhere in an expression, but the fact that functions put their return value in a call message, instead of passing it on a stack as in most languages, gives us the solution. All function calls in an expression will be evaluated before the expression, and then the entire expression can be calculated at once. For example, given the following statement:

```
i = j*7 + f(k);
```

most languages would compile it into the following:

```
load j; load 7; mult; load k; call f; add; store i;
```

but for us it would be:

```
load k; store temp_call_message.k; call f(temp_call_message);
load j; load 7; mult; load temp_call_message.result; add;
store i;
```

The additional store and load of the call message are required because we don't know beforehand what processor the f process is running on, so parameters must be transferred through shared memory, and not the processor's local stack. The rule is that the compiler can't generate any code for an expression until it is ready to generate the code that uses up the expression. This way the stack is only in use for short periods of uninterruptible calculation.

To implement this, the compiler parses the expression without generating any code. When a function call is encountered, code for it is generated immediately, and a reference to the result is remembered in place of the function call. Finally, at the end of the expression, code for the expression is generated. This means that nothing is put on the stack until ALL of the data to calculate the expression is ready. Immediately following the expression a store instruction is generated to remove the result from the stack.

The parameters for function calls are moved into call messages that are temporarily allocated by the compiler. The call message is used from the time the function code is generated to the time the function's result is used. For example,

```
i = f(x) + f(y) + f(z);
```

requires three call messages because the functions' results are not used until they ALL have been calculated. On the other hand,

```
i = f(f(f(x)));
```

only needs one call message that can be reused for all three calls.

Most NIL operations have an equivalent IL opcode. The process of compiling them just involves sending out the opcode followed by the offsets describing the operands. These instructions are: ACCEPT, CALL, COMPLETE (compiled as WAIT), CONNECT, CREATE, DELETE, DISCARD, FORWARD, MOVE, PUBLISH, RECEIVE, REMAP, RETURN, SEND, and UNPUBLISH. A few like CALL can have expressions for parameters. These expressions are just moved into temporary variables, and then the statement is generated using the temporaries instead of the expressions.

Of the control instructions IF and LEAVE are quite straight forward to compile. They are done with the branch and conditional branch opcodes.

LOSEing a row is the same as DISCARDing it.

The ALLOCATE statement in (CLUMPS) NIL is not needed for messages due to the lazy allocation scheme (see interpreter implementation below). For

rows, there is an allocation opcode for every case, detailed above in the IL section. Similarly all variants of the FIND statement have opcodes except for FIND .. WHERE. To compile:

```
FIND row WHERE(boolean_expression);
```

we use:

```
   LOSE row;
A: FIND row AFTER row;
   IF NOT boolean_expression THEN GOTO A;
```

Notice that to FIND the first row we just LOSE the row and then find the one after itself (i.e. the first row is after an empty one). Similarly to find the last row we look for the row before an empty one.

The rest of the instructions require a bit more effort to compile, and will be explained one at a time.

4.2.2.1 Assignment (=)

The assignment operation copies the value of an expression to another variable. This is different from MOVE which transfers data from one name to another. After a MOVE the source is uninitialized. Assignment is only valid for the types Boolean, Char, Integer, Long, Strings, Secondaries, and Rows. Assignment is done by calculating the expression on the stack, and then storing the result in the destination.

String assignment is more complicated than scalar assignment, especially as indexed and substring assignment is possible. Strings are represented on the stack as an address and a length. This allows constants and indexed strings to be easily used, and single characters (or integers) can be treated as strings. Operations that create a new string out of one or two strings (such as concatenation) expect the source strings' addresses and lengths on the stack, and create a new string in a temporary variable. The address and length of this temporary is left on

the stack so it can be used for the next calculation. For example the IL for this NIL statement:

```
s = s ! "abc" ! s[2;i];
```

is:

```
        ldo     s         ;get add and length of s on stack
        ldcs    L1        ;get add and length of "abc" on stack
        sconcat temp1     ;concatenate and store in temp1
        ldo     temp1     ;get add, length of temp1
        ldo     s         ;get add, length of s on stack
        ld2               ;get low index on stack
        ldo     i         ;get length of desired substring
        substr            ;modify address, length by indices
        sconcat temp1     ;concat past two results -> temp1
        move    temp1,s   ;store result in s
        ...
L1:     3,"abc"
```

For example the following kinds of statements are valid, given c as a character, s as a character string, and i,j as integers:

```
s="abc";
s=s[3;*]!c;
s=c;
s[10]="abcd"[i];
s[i;3]=c!s[j;2];
s[2;2]="abc";
```

When the statement calls for assignment to a substring as in s[5]='a' or s[i;*]="abc" the target string MUST be the same length as the string expression. If so, no new memory is needed.

4.2.2.2 REPEAT

Most variations of the REPEAT statement are easy to compile with branches and conditional branches. See [16] for details. Repeating over

---

16. Draft NIL Reference Manual, RC 9732, IBM T. J. Watson Research Center, Yorktown Heights, NY, Dec. 1982, pg. 95

row objects is a bit more tricky. Given:

```
REPEAT FOR (row) WHERE (boolean_expression)
  {loop statements}
END REPEAT
```

we compile:

```
   LOSE row;
   DO
A:    FIND row AFTER row;
   ON (NOT FOUND) goto B;
   END DO;
   IF NOT boolean_expression GOTO A;
   {loop statements}
   GOTO A;
B:
```

This is very similar to a repeated FIND..WHERE, except that when a row finally can't be found that meets the specification, the repeat loop is terminated normally.

4.2.2.3 SELECT

The SELECT statement is used to transfer control to one of a group of options. It is identical in function to Pascal's CASE statement. Its syntax is straightforward:

```
SELECT (expression)
   WHEN (expression1) statements
   WHEN (expression2) statements
   ...
   OTHERWISE statements
END SELECT
```

If the main expression matches the expression in a when clause, the statements following that WHEN are executed. SELECT statements are

compiled into the equivalent IF THEN ELSE statements.

There is a version of the SELECT statement used to wait for external events. For example:

```
SELECT WAIT
   EVENT (object1) statements
   EVENT (object2) statements
   ...
END SELECT
```

The objects can be either components or primaries. The event being waited for is the completion of a component, or the arrival of a message at a primary. The given syntax is a slight modification from the official syntax. In the original each event had a guard, i.e. a boolean expression that could be used to disable a particular event for a particular execution of the SELECT WAIT statement.

If a SELECT WAIT statement has only a single EVENT then it is compiled simply into:

```
WAIT object1
statements
```

If it has more than one EVENT, it is compiled into a loop. For example, if all object are primaries (queues):

```
REPEAT
   IF LENGTH(object1) > 0 THEN
      statements
      LEAVE REPEAT
   END IF
   IF LENGTH(object2) > 0 THEN
      statements
      LEAVE REPEAT
   END IF
   ...
```

```
    SLEEP
END REPEAT
```

Note that LENGTH(object) > 0 is replaced by COMPLETE(object) if the object is a component. This is much less efficient that the single EVENT case, but it avoids all of the problems associated with setting up and disabling triggers on multiple objects.

4.2.2.4 Exception Handling

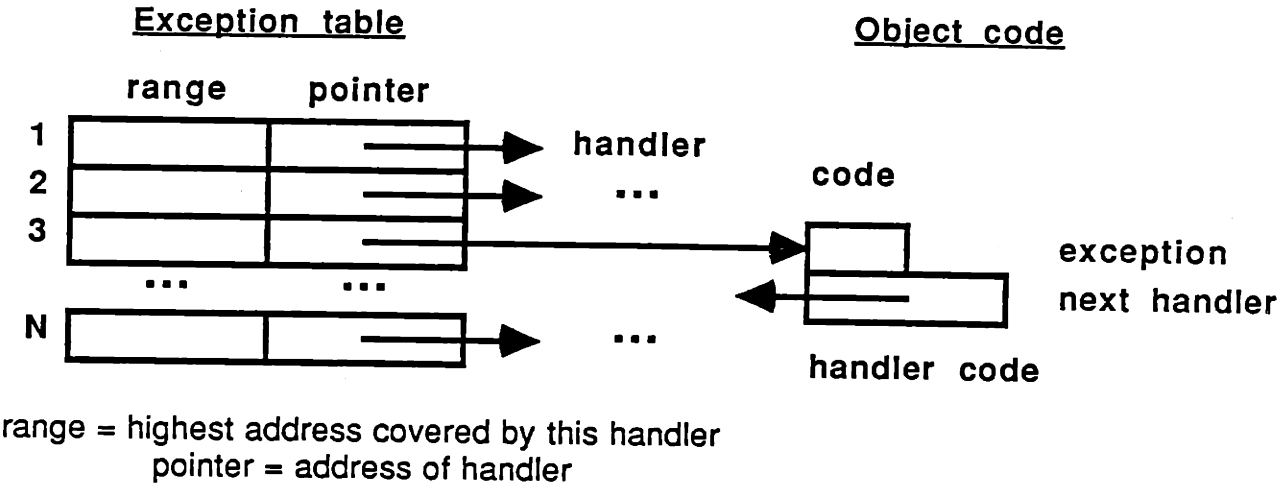
The DO...ON exception...END DO constructs are used to compile an exception handler table that contains pointers to the appropriate error handler for the indicated section of the program. A LEAVE DO is inserted just before the ON, so that normal program flow skips the exception handler.

The exception handler table lists the innermost exception handler for each range of code addresses. An entry is made in this table for each DO statement and each ON statement, since these statements determine which exception handler is currently in effect.

When an exception occurs, the value of the relative program counter is used to search the exception handler table (see figure 5). If its value is between zero and RANGE1, the interpreter will try to find the handler for the exception in HANDLER1. If its value is between RANGE1+1 and RANGE2, the interpreter will try to find the handler in HANDLER2 and so on. RANGEn must point to the last executable byte. If the handler does not handle the exception that occurred, its parent (i.e. the surrounding exception handler) is examined. The compiler always creates an exception handler that surrounds the entire program, handles all exceptions, and just stops execution.

The handlers themselves are mingled with the code, and consist of a single byte describing the exception being handled, and a pointer to the next enclosing handler. These three bytes are followed by the actual code of the handler.

Figure 5: Exception Handlers



4.2.3 Assembler

We use a very simple assembler for the intermediate language. The input is just a list of expressions and symbol definitions. The first pass just evaluates the symbol definitions and creates a symbol table. Object code is generated during the second pass.

The assembler is implemented with the same compiler generator as the compiler itself. The source syntax is given in Appendix C. During the first pass the assembler assigns values to symbols. This can be done with an equate statement of the form: identifier EQU expression. The expression should not contain any forward or circular references. Symbols can also be defined like labels by following them with a colon, e.g. START:. This sets the symbol to the current value of the object code counter. This counter is initialized to zero at the start, and increments by one for every byte of object code produced.

Expressions can contain symbols, character or integer constants, the period symbol (.) which represents the current value of the object code counter, and the operators +,-,*,/,MOD,&,|,XOR,>> (shift right), << (shift left),~ (not). Expressions can be arbitrarily parenthesized. All

operations are performed on 16 bit integers.

Object code is produced during the second pass. If an expression is not part of an equate statement it is output as object code. If it is followed by an at sign (@) it is output as a two byte integer. Otherwise only one byte is produced. Expressions can be separated by commas or semicolons, but this is not necessary. Character string constants can also be turned into object code. The intermediate language opcodes are handled just like other symbols, and are read in from an initialization file before the first pass.

Figure 6 contains an example of how the compiler and assembler would deal with a short process that finds the maximum of two integers.

Figure 6: Compiler/Assembler Example

```
threeint IS CALL MESSAGE (i,j,result: INTEGER) RETURNS(cmp:COMPONENT)

max: PROCESS (input: threeint)
     CREATE PHASE
        IF input.i > input.j THEN input.result=input.i
        ELSE input.result=input.j
        END IF
END PROCESS
```

The assembly language source for this program is:

```
V1 @, EXCEPTIONS @,      {See component in figure 4 for
                          the layout of a code block}
   ldo V1_input, V3_j,
   ldo V1_input, V3_i,
   lt                    {do the comparison}
   bz L2 @,
   ldo V1_input, V3_i,
   sto V1_input, V3_result,
   bra L1 @,
L2:                      {else clause}
   ldo V1_input, V3_j,
   sto V1_input, V3_result,
L1:
   ret V1_input, 0, $0, {return the result}
```

```
   stop

L3: $0, L3 @,
   stop

EXCEPTIONS:                (exception handler table)
    L3 @, L3 @,


V1:                        (the top-level variables)
    $F9, 0, V1_-V1 @,
V1_input = .-V1
    V3 @,
V1_:

V3:                        (the structure of the call message)
    $F9, $3D, V3_-V3 @,
V3__RETURN = .-V3
    $F2, 0,
V3_cmp = .-V3
    $F7, 0,
V3_i = .-V3
    $F2, 0,
V3_j = .-V3
    $F2, 0,
V3_result = .-V3
    $F2, 0,
V3_:

END
```

4.2.4 Interpreter

Each processor in a CLUMPS node executes its instructions directly from a local ROM. The ROM routines perform three tasks; first, all initialization of a CLUMPS node (communication, memory management, self-test, catalog, task list); second, all the communication tasks for that branch; and third, it shares responsibility for executing the node's operating system and NIL processes.

Most of the communication work (sending and receiving bytes) is handled via interrupts. The rest (allocating buffers, changing parameters) is accomplished during time yielded by the interpreter. Whenever the interpreter starts an instruction with an empty stack it checks to see it the communication process needs any time, and if so, jumps to it.

Any leftover time is spent on the interpreter. A NIL program is just a set of processes. Each process is represented by a memory structure called a shared memory process control block (SMPCB, diagrammed above under component). This block contains pointers to other processes, the code block, the process's parent, the current program counter, and the top level variables.

Processes can be in one of three states: running, waiting, or sleeping. Since the number of processes can be greater than the number of processors, active processes are kept on a queue until a processor becomes available. These are "sleeping". A processor will become available when the process it's currently executing finishes, goes to sleep (either by explicit call, or being swapped out after executing a fixed number of instructions), or starts waiting. When a process is waiting for an event (a call to return, a message to come in, or a subprocess to complete) it is not kept on the active list, but is replaced there by the process that causes the event to occur.

4.2.4.1 REMAP

The REMAP operation of NIL converts between objects (stored in their internal form) and strings (stored as a two byte length followed by that number of bytes). This allows objects to be transferred from one node to another, even if they have a complicated structure. First the object would be converted to a string, then the string is sent to another node where it can be converted back to a object.

Primaries, secondaries, components and rows cannot be REMAPed. If an object contains any of these they will be ignored. The interpreter constructs a string from a object very simply: it examines all of the items in the object in order, copying their value into the target string. Contained objects are likewise enumerated as they occur. To remap from a string to a object, the object's descriptor is used as a guide in parsing the data.

4.2.4.2 Lazy Memory Allocation

Many of the changes and simplifications we have made to NIL are the result of having an interpreter that can handle many of the memory allocation problems previously solved by the compiler or the programmer. The standard NIL compiler goes through a lot of trouble (see "Typestate checking" in [16] pg. 2 and chapter 3) to ensure that the messages, rows, etc. are allocated before they are used. If this checking were not done, it would be possible to use uninitialized pointers and cause bugs that are very difficult to pinpoint.

Since (CLUMPS) NIL is interpreted, the interpreter can notice when one is about to use an unallocated variable, and allocate it on the spot. It can also notice that one is assigning to an allocated variable, and discard the old data first.

The rest of what the interpreter does is self evident. It moves through the intermediate language object code, executing one instruction at a time. When a process it is executing finishes, waits, or goes to sleep, it gets another from the active queue, and runs it.

Chapter 5

OPERATING SYSTEM

"The Operating System" is the name given to all of the system software that is not part of the NIL compiler/interpreter. Like the NIL software, it has pieces all over the system: in the branch ROMs, in the host, and in global memory. The primary function of the operating system is to connect the user's NIL processes with each other (across nodes), and with the host. The operating system also performs some housekeeping such as system initialization and loading processes.

5.1 Communication Facility

At the heart of the operating system is the node to node communication facility. This part of the operating system is a trade off between the flexibility of a low level interface, and the simplicity and speed of a hard coded high level interface. We wished the user to have the flexibility to create his own retransmission scheme, but not be forced to deal with byte level issues such as CRC calculation.

The first decision was to have a packet level interface. The transmitter designates a link number and a character string, and the receiver gets it, or gets a notification of an erroneous attempt. All the detail of headers, buffers, CRC checking, etc. is done by the operating system. It is unlikely that the interpreted NIL code could keep up with a byte level interface. NIL's REMAP operation makes it especially easy to deal with packets as character strings.

The user can set the line characteristics of baud rate, error rate, delay and carrier sense delay (in the case of radio). The user is responsible for having nodes at both ends of a link set the same baud rate (which can range from 0 to 76.8 Kbit/sec).

The error rate is specified in errors/byte, but is calculated a packet at a time. The packet length in bytes, L, is multiplied by the error rate R, and this probability is used with a pseudorandom number generator to determine if the packet will have an error or not. This is not the same as the probability of error if all byte errors are independent:

$$\text{Packet error prob} = 1-(1-R)^{L}$$

However this is very close to LR for most practical cases when LR is small and it is easier to calculate[17]. Note that errors are generated in the receiver, and not the transmitter, so that in the case of radio all receivers will not necessarily encounter the same errors due to pseudorandom noise.

The user can also specify a line delay, which is implemented by having the receiver wait after receiving a packet. Delay is not particularly useful in simulating radio networks, because it would appear the same to all receivers, which is not realistic. (See the discussion above in the section on Modelling.)

Since each link is bidirectional, it will be in one of four states: transmitting or not, and receiving or not. These states can be tested at any time by a program. While simulating radio communication, one often wants to use a carrier sense protocol. This just means that the line is not allowed to go into the state of transmitting and receiving at the same time. Carrier sense mode can be specified upon initialization, and it

---

17. Bit errors on most communication channels are not independent [Tanenbaum, Computer Networks, "The Nature of Transmission Errors," pg. 125], but occur in groups. If the error groups are smaller than the packet size, but occur less frequently than packets, and independent of each other, then this approximation is quite good if we divide the byte error rate by the average group size.

prevents a packet from being sent while something is being received. Note that the delay timing starts after the decision to transmit has been made.

The last branch communication parameter is wire or radio mode. In radio mode, reception is disabled during transmission. The transmission period is considered the time from when the transmission decision is made, until after the inter-packet gap after the transmitted packet.

The network consists of links used by the operating system and links used by the experiment. The operating system links are used to communicate with the host. They are configured into a spanning tree. After power on or reset, the first message to come in over any link is considered to be the operating system, and it is installed as a process and run. The node that sent it is called the receiving node's parent (which may be the host).

In a node with n links (numbered 0 through n-1) the link to the parent and all higher numbered links are considered to be the operating system links. All lower numbered links are the experimental (i.e. user) links. Since the operating system links form a spanning tree with the host at the root, all messages for the host are simply sent to the parent node.

5.1.1 NIL interface

The operating system provides a number of services accessible to NIL programs. Some, like the CONNECT and CREATE operations, are defined as part of the NIL language itself. The services that are specific to the (CLUMPS) system are listed below.

5.1.1.1 TIME

The clock in (CLUMPS) consists of a 48 bit counter that increments every 406.9 ns. It can be used to measure the time between events. The clocks on all of the nodes are synchronized by the operating system when it is initialized. The built in variable time returns the middle 32 bits of the clock as a long integer. This function counts every 104.2

microseconds, and rolls over after 124 hours. The time can be set with an assignment statement. All communication packets are timestamped when they are received. The clock routine is totally internal to the interpreter.

5.1.1.2 READ, LOG

Since (CLUMPS) nodes have no file storage, the operating system provides file service using the host. This is a very simple system that just allows character strings to be stored and retrieved. There are two functions: read and log.

Read takes a filename as a parameter and returns the contents of that file (from the host) as a character string. Log takes a character string and appends it to the host's log file.

The file functions are not part of the interpreter, and must be linked with a CONNECT statement before being used. The following example shows the code needed to read a file called "fred" and write it into the log file. (The NIL keywords are capitalized for emphasis.)

```
readmess IS CALL MESSAGE (          {These definitions are global,}
     data,result: CHAR STRING);     {so they doesn't have to be   }
logmess IS MESSAGE (                {declared every time.         }
     data: CHAR STRING);

DECLARE
   read: readmess CALLPORT;
   logp: logmess SENDPORT;
   logm: logmess;
          ...
CONNECT read TO "read";
CONNECT logp TO "log";
CALL read("fred",logm.data);
SEND logm TO logp;
```

5.1.1.3 Communication Interface

Data and initialization messages are sent to the communication system using the NIL instructions COMCALL and COMSEND. These correspond to the standard CALL (synchronous) and SEND (asynchronous) instructions, but instead of specifying a secondary for the messages to be sent to, they are sent directly to the appropriate branch. Only communication messages can be sent with these instructions.

Communication messages come in two types: cominitmess and commess. The former is used to initialize a communication line, and the latter is used to transmit data over the line. A line is initialized with the following parameters:

branch number	Branch to initialize
baud rate	Baud rate (see below)
error rate	Errors per 2**24 bytes (for receiver)
delay	delay in ticks (see below)
carrier sense	delay in 16ths of a bit (see below)
receive	Callport to get received messages
archive	Callport to hold messages after xmit

The baud rate is specified in internal units that can be set by the NIL function BAUDRATE() which takes the standard rate in bits/second, and returns the internal units. See section D.6 for an example.

The error rate is specified in errors per 2**24 bytes. The parameter is an unsigned 16 bit number. This means that the largest error rate is one per 256 bytes. Of course, the way this is calculated, with that error rate a 257 byte long packet would always have an error. If the user wants a more realistic pseudorandom error rate he can always set this one to zero, and build his own into the lowest level of his software.

The delay is specified in the same units as the TIME variable. It is the amount of time that the system waits before passing on received messages to the user.

The carrier sense delay specifies one of three modes. If it is zero then we are in full duplex wire mode. If it is negative then we are in

radio mode without carrier sensing (i.e. half duplex, transmitting takes precedence over receiving). If it is positive it specifies the carrier sense decision time in 16ths of a bit.

The receive callport should be connected to a queue where the user wants the system to puts messages received from this line. The archive port is where the system returns data messages after they have been transmitted, if COMSEND was used. If COMCALL was used they are returned to the caller after transmission.

The message format that is used for communication data has the following structure:

```
commess IS CALL MESSAGE
   (flag: CHAR;          {after operation: 0=OK, other=error}
    branch: CHAR;        {branch number (0-7)}
    timestamp: LONG;     {time at the end of the com operation}
    data: CHAR STRING);{the data}
```

5.1.1.4 COMSTATE

At any given time a communication line can be transmitting or not, and receiving or not. A program can test this with the COMSTATE(branch) function call. It returns a number 0 through 3 based on the state of the given branch as follows: 0-idle, 1-receiving, 2-transmitting, 3-both.

5.1.2 User interface

The host is used as the operator's terminal during simulation runs. These runs are carried out in phases as follows:

1. The switch is set to the desired network topology.

2. A command is given to load the operating system into all of the nodes.

3. The system asks for the main user task which is sent to it automatically. It can start other processes and run the simulation.

A more detailed description of this process can be found in Appendix C.

5.2 Communication Implementation

One of the most challenging parts of the implementation was that of the communication subsystem. It has to handle a variety of asynchronous events. From the communication hardware can come interrupts for byte reception, transmit buffer empty, receive error, and timeout. Bytes must be accumulated into packets which means dealing with the (CLUMPS) memory management routines, the clock routines, and compensating for errors on multiaccess lines. Requests from processes for status and initialization must be handled.

Our general strategy to deal with all of these issues was to divide the communication tasks into two parts: the quick tasks that must be handled immediately such as byte level reception, and the others that are not guaranteed to take a fixed amount of time such as memory allocation. The first group would be handled by interrupt routines that just deal with the communication hardware and memory buffers; when something complicated happens (such as a packet boundary) the main communication routine is alerted. This routine alternates with the interpreter for use of a branch's microprocessor. Before the interpreter starts to process a new statement it checks if the communication routine needs any time. If so, communication gets priority.

5.2.1 Packet Buffering

One complicated issue is how and where to buffer packets as they arrive. Remember that individual processors have only 2K bytes of local memory while the node has 64K bytes of shared memory. Some local memory must be used for the interpreter's stack and internal variables. If packets are buffered into local memory until they are complete then they would have a maximum length of only 1K bytes. Packets in communication networks are typically 128 to 256 bytes, so a 1K limit isn't severe.

The main problem with first buffering a packet in local memory is that by the time a global memory block is allocated for the packet a new packet may be arriving and there would be no place to put it. Efficiency would also be reduced by moving data twice: first into local memory, and later into global memory.

Our approach to buffering will be that local memory is only used at the beginning of a packet, while a global block is being allocated. Thereafter data will be stored directly in global memory. Packets will be preceded by their length so that the block can be allocated precisely.

5.2.2 Error Handling

Packet errors from the user's point of view can come from three sources: errors introduced by the channel's simulated error rate, errors due to packet collision on multiaccess channels, and "real" errors due to a failure or noise in the (CLUMPS) system. All of these errors will be reported in the same way by the operating system. If the system is suspected of making "real" errors, it can be tested by setting the simulated error rate to zero.

Packets are structured on the channel as follows:

packet length (2 bytes)	-- data --	xor checksum (1 byte)	-- inter packet gap

Packets are separated by a gap (idle channel) of about five bytes. All bytes are sent with parity. An error is reported if the packet length is wrong, a channel error occurs, a parity error occurs, the checksum is wrong, or it is time for a induced error. In case of error, the data (if any) is discarded.

Chapter 6

PACKET RADIO

The issues involved in setting up a complete software system for packet radio experimentation, and proposed protocols are discussed in this section. A simple sample protocol is presented to illustrate the software implementation of a layered protocol.

6.1 Structure of an Experiment

What is needed is a base (or library) of software that implements a simple data communication network. This base can then be extended to examine different areas of concern. In addition to simulation specific programs for system configuration, packet generation and performance measurement, we will want modules that implement simple protocols for the data link, network and transport layers of the ISO standard model[18]. Some sample modules have been implemented as part of the example program described in the next section, while others are merely discussed. Except for a network configuration program (which would run on the host) each of the areas discussed below would be implemented as one or more separate NIL processes.

6.1.1 Network Configuration

The first step of a simulation is to choose the network topology and

---

18. Andrew S. Tanenbaum, Computer Networks, Prentice-Hall, 1981, pg. 15

set it with the switch. While this is currently done by manually programming the switch from a terminal, for larger networks it would be simple enough to have the host set the switch from a configuration file. This configuration could either be specified by hand, or created randomly by a utility program.

6.1.2 Data Link Layer

The data link layer is responsible for reliably transmitting data from a node to a neighboring node. This is commonly accomplished by sending data in blocks called frames[19] that are separated by special bit sequences called flags. A protocol (termed ARQ for Automatic Repeat reQuest) is then added to the frames to ensure the reliability of this layer by retransmitting frames that are not properly received and acknowledged.

Having an ARQ protocol implies a certain amount of addressing when dealing with multiaccess channels since it is not practical to require an acknowledgement from every radio transceiver in range of a broadcast message. By considering the radio channel to be a collection of links between pairs of nodes with the restriction that certain links cannot be used simultaneously we can implement the standard ARQ protocols over these links. These include go back n and selective repeat protocols[18 pg. 155].

In (CLUMPS) the lowest half of this layer (framing and error detection) is built into the operating system (see previous chapter). Inter-packet gaps (idle time) are inserted during transmission to allow receivers to frame packets. Packets are transmitted with byte parity and a checksum to detect most errors. Note that this level does not provide any addressing (i.e. radio connections will be broadcast only).

---

19. In this thesis we often use the term packet when we really mean frame. The units of data exchanged at the data link layer are frames, while those at the network layer are packets. They are not necessarily the same size. Since (CLUMPS) supports variable length frames it is often natural to make them identical.

The ARQ protocol is left up to the user. This split relieves the users from the byte oriented tedium of checksums, but leaves him free to experiment with various ARQ methods.

6.1.3 Network Layer

The network layer enables packets to travel many hops to get from any node in the network to any other. This layer is responsible for both how a packet gets to its destination (routing) and when it gets there (flow control). Network layer protocols are the most challenging because they must control a group of independent devices to form a coherent system. Since most experimentation will be in this area, we just provide the sample described in the next section.

6.1.4 Transport Layer

The transport layer is responsible for orderly, error free end to end communication. Just as the data link layer's ARQ protocol eliminates link errors, the transport layer handles node errors. It deals with out of order packets, and lost or duplicated packets. One easy implementation would be to provide a go back n protocol between the source and destination nodes. Protocols for this layer and higher ones can be shared by wire and radio networks.

6.1.5 Higher Layers

These layers are generally application specific, so they are mostly left up to the user. We do provide some packet generation routines that blindly source and sink packets. Of course, the user is free not to use these routines, and program his own higher levels.

The packet generation routines simulate computer traffic. This may be the place where our simulation strays the furthest from actual network conditions. In the case of proposed networks our models of average and worst case loads will be mostly guesswork.

A large variety of packet generation scenarios are possible. Packets could be created independently from the congestion in the network, as in most file transfer or mail applications; they could be created as a response to received messages as in transaction processing applications; or some combination of the two as in remote terminal applications. (CLUMPS) provides evenly distributed random numbers from which many different interarrival time distributions and packet size distributions can be synthesized.

6.1.6 Measurement

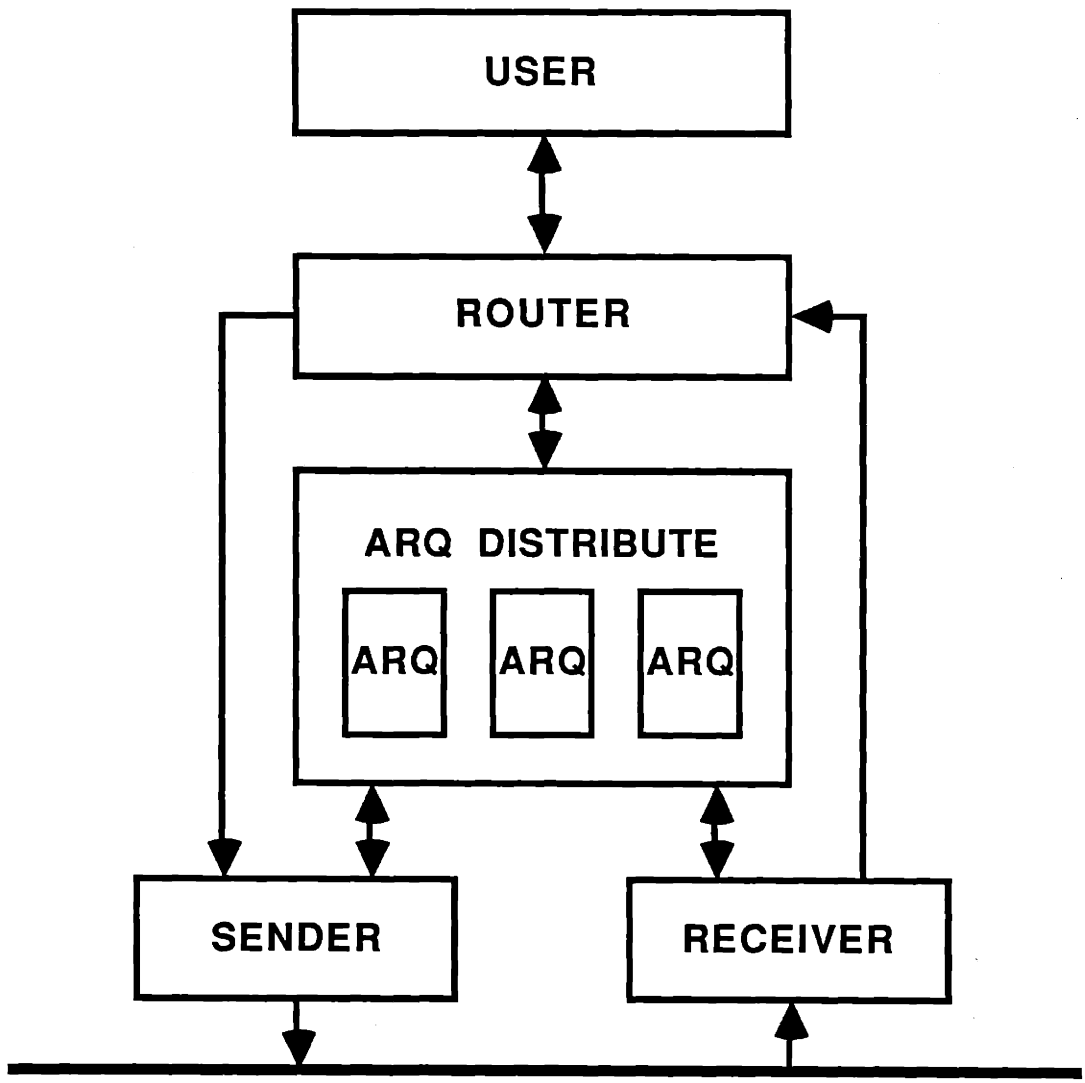
Delay can be measured by the packet generation layer. Delay can be determined by placing the time in a packet when it is generated, and subtracting it from the current time when it is received. Accurate individual delays require all nodes to have synchronized clocks, but if equal numbers of packets are sent between all node pairs then the total delay is just the sum of the individual delays regardless of the initial value of the clocks.

Other parameters are best left up to the user since different experiments are looking for different things. The operating system does provide information to the user on the time of packet reception and transmission, and on the nature of reception errors.

6.2 Radio Simulation Example

The program described below serves to illustrate the pieces necessary to run a packet radio simulation on (CLUMPS). The program itself is listed in Appendix D. See figure 7 for an outline of the program's processes and how they interrelate. The program communication with two kinds of packets: broadcast packets with routing information, and data packets that are intended for a particular node.

Figure 7:
Packet Radio Simulation Example



6.2.1 Data Link Layer

At the bottom there are two routines sender and receiver. Sender accepts packets to be transmitted, and sends them with carrier sensing. If the carrier is on, sender waits a random time and tries again. After a successful transmission the packet is returned to the higher layer.

Receiver accepts packets from the airwaves, discards erroneous packets, sends broadcast packets to the router, and sends data packets to the ARQ process.

The ARQ level maintains a separate process for every logical link (i.e. one for each neighbor) and one process to dispatch packets to the others based on the address in the packet.

We use a simple stop and wait ARQ algorithm (a version of go back n). This adds two control bytes to packets, and occasionally generates packets with only control information.

6.2.2 Network Layer (Routing)

The sample program does shortest hop routing. It is a subset the original ARPANET routing algorithm[20] with the additional assumption that there will not be any permanent node or link failures. It creates a table of destination node numbers, their corresponding intermediate node numbers, and their distance in hops.

Broadcast messages are used to convey routing information to neighbors. Every node periodically broadcasts the distance it is from all other nodes. When a broadcast message is received, its data is compared to our stored information to see if the neighbor who sent it is more than one hop closer to any node than we are currently. If so, we alter our path to that node to go through the neighbor. If the routing table is

---

20. J.M. McQuillan and D.C. Walden, The ARPANET design decisions, Comput. Networks, vol. 1, Aug. 1977

changed by this process we broadcast the new table to update our other neighbors. This method ensures that eventually every node will know how to get to every other node in the fewest number of hops. If there are several paths of equal length only one will be used, however.

Using this module adds four bytes to a packet: immediate destination, source, final destination, and original source. Packets received at a given node without an immediate destination equal to that node's ID or zero (broadcast) are discarded. If the destination is the same as the immediate destination, the packet is passed on to the higher protocol layers. Otherwise the immediate destination is changed according to the routing table, and the packet is queued for transmission. Note that this layer steals the broadcast capability for its own use; it is not available to the user.

6.2.3 Higher Layers

The transport protocol is just another stop and wait, but without retransmission since we assume that there will not be any node errors. Each node sends a packet to every other node. When a packet is received from a node another one is sent to that node. The simulation is halted after eight packets are received from each other node.

6.2.4 Network Operation

Appendix C contains step-by-step instructions for running experiments on (CLUMPS). The general procedure is as follows:

1. Reset: If network was previously initialized, the host sends reset commands to all children, who reset their children, then themselves.

2. Choose network: either randomly or from a user configuration file. The configuration is currently entered from a separate terminal, but this process can be automated.

3. Initialize node(s): A special initialization packet containing the

operating system is sent to the node connected to the host. Upon receipt the operating system is started. It sends a copy of itself through all of its operating system links. Therefore the operating system is propagated to all nodes in the network.

4. Send user program to all nodes: The operating system in each node waits a few seconds for the system to initialize, then sends a request to its parent for the user's program. This request is processed like all other file read requests. The user program is installed in all nodes as a process, and run. It can then start other processes, take control of the user links, and start the experiment.

6.3 Proposed New Protocols

The communication routines described above are adequate to construct a working model of a wire or radio communication scheme. Once they are working we can experiment with other, innovative algorithms. In particular we propose an improvement on the data link layer used with radio networks.

As noted earlier, the major problem not solved by carrier sensing is one of indirect collisions, i.e. two or more nonadjacent nodes transmitting simultaneously to a common receiver. The simplest improvement to deal with indirect collisions would be the addition of long random waits between transmissions. Sufficiently long waiting periods would improve the chances that all neighbors of one's intended receiver are quiet during transmission. Further improvement may be possible by choosing waiting times according to estimates of network congestion.

We also want to investigate a still more sophisticated scheme that uses short reservation packets to allocate a larger portion of the system's space-time resource for the transmission of a data packet. It would work as follows: When a node desires to transmit a data packet and is free to transmit, it sends a short request to the receiver. If the receiver

receives the request and is free to acknowledge it, it replies with a short "go ahead" message. The transmitter then sends the data. If properly received, the receiver immediately sends an acknowledgement, or another request. Any third node transmitting a request or hearing a request, a "go ahead", or a collision is required to wait until the transmission is over (or a maximum packet transmission time) before becoming free to transmit again.

It can be shown that, by using the above scheme, data packets will be subject to collisions only if reservation messages suffer direct collisions. The proof is quite simple. If any data packet did suffer a collision that means that the node (X) causing the collision didn't receive the go ahead message. X would not receive the go ahead message only if it was transmitting, or if the go ahead message caused a collision. In the latter case the collision forces X to wait and not transmit. If X was transmitting then carrier sensing would prevent the receiver from sending the go ahead message in the first place, unless the go ahead message exactly overlapped the message that X was sending (i.e. a direct collision). Note that the better carrier sensing works, the less chance there is of direct collisions.

The hope is that collisions will be reduced, and throughput improved due to the fact that reservation packets will be much shorter than data packets, and hence less likely to collide. Questions to be answered by simulation include whether the overhead of reservations is justified, and how the transmission parameters (maximum packet size, waiting time between failed requests, etc.) can best be chosen.

Chapter 7

CONCLUSIONS

(CLUMPS) is an attempt to simplify the process of designing and optimizing communication systems. Time will tell how successful it can be. After a few years of work on its design and implementation the natural question is what would be done differently if we started today? Is the concept sound? Could there be improvements?

7.1 Alternatives

Throughout the thesis we have discussed alternatives to the detailed design decisions that were made. In this section we examine the overall system and see how the design might be radically different if we were starting in 1986 rather than 1983.

Technically speaking, the goals and design philosophy described here are still very valid. The idea of doing real time, distributed simulation of communication systems is a good one. No other method can guarantee the same accuracy and speed.

One thing that has changed a great deal since the project was started is the economics of microcomputer systems. While the actual cost of a (CLUMPS) node probably hasn't changed much in the past few years (about $1000 per node) the cost of a commercial microcomputer has plummeted due to the great volumes involved. Doing a completely custom design like (CLUMPS) does not make sense today. A better alternative would be to use personal computers outfitted with several serial communication lines as the network nodes. The hardware cost would be about the same, but there

would be huge savings in development time both for hardware (only the switch would have to be designed) and for software. Commercial operating systems and compilers could be enhanced for our application, rather than being written from scratch.

Another important aspect of system design, one that I have unfortunately ignored, is management. There are two fundamental changes that intelligent management alone would have required: vertical design and standard software.

Vertical design means giving individuals as much autonomy and responsibility as possible over an identifiable portion of the project. The idea is to reduce the communication necessary between project members, and give them the greatest possible sense of accomplishment. This method does not just affect how a project might be divided, but should be a consideration on the design that is selected. For us that might have meant having a design with a computation processor and a communication processor, rather than many processors that do everything. Instead of having hardware people and software people that must interact on many levels, we might have had one person responsible for the entire communication subsystem hardware and software, and another couple on the computation subsystem. In any case, these kinds of issues were not considered during the design of (CLUMPS).

While using NIL as the system language was a good choice technically, it was not really very practical managerially. It caused us to have to start entirely from scratch, adding much time to the project, time that could have been spent experimenting with communication systems. A better choice for an interpreted language would probably have been a Pascal P-code, or Forth implementation. Either would need extensions for communication, but they would have allowed us to get something working quickly, rather than waiting for the entire system to use a part of it. Both Pascal and Forth implementations are widely available, and are well known. I still am unsure of whether using NIL was the best overall choice.

After adding these economic and managerial factors to the technical

ones, the system we might design today would look quite different than (CLUMPS). It would still be a network of computer nodes connected by a digital switch. The nodes would be standard personal computers with additional serial communication ports. These ports could either be controlled by their own communication processor or they could share the main processor, giving us a cost/performance trade-off. The nodes would use their standard operating system and compilers with some additional libraries of routines to interface to the communication facilities.

While both the communication capabilities and multitasking capabilities of this new system would not equal that of (CLUMPS) (due to fewer processors and less specialized software), there would be great savings in development time to get the system working. Training of future users would also be much easier since they would be using standard systems and languages.

7.2 Improvements

We are not starting from scratch however, and given that we have such a powerful custom system are there any areas that could be improved? As (CLUMPS) gets used for designing communication networks and protocols many more possible enhancements will be become apparent.

What follows is a short list of suggestions that are either the result of our experiences with the system or were not considered pressing enough to implement in the first pass to get a working system. Often we suggest ways to simplify NIL's semantics and give the user more power. The funny thing about engineering is that sometimes it's easier to make something complicated than to make it simple. Included in each suggestion is an estimate of the amount of work involved to implement it. The categories are: minor: less than a day, average: less than a week, and major: more than two weeks.

7.2.1 Communication

- Allow user to specify buffer size like other communication parameters (baudrate, etc.). This buffer would just be used for incoming packets, and could be used to emulate systems with limited buffer space. After deciding the best way to interface this with the operating system it is a minor change.

- More accurate timing of inter-packet gaps. Currently the system uses a free running counter to determine the gaps. If two counts go by without a reception then the receiver knows it is between packets. The transmitter waits at least four counts between packets. The problem is that because bytes are not synchronized with the counts there is a fair amount of uncertainty between the receiver's threshold and the transmitter's idle time. This means that we have set the gap longer than we might otherwise like to ensure that succeeding packets are not accidentally merged. The gap is currently about 5 bytes long.

  The solution is either to go to a bit stuffing method and separate packets with flags, to signal with a line break condition after the last byte is transmitted, or to more accurately time byte receptions and transmissions. Since our hardware is not suited for the first solution, and the second solution might confuse packet gaps with transmission errors, the last solution has the most appeal. This an average to major undertaking, and not a very urgent one.

7.2.2 NIL

- Add integer exponentiation and square root operations. This would make it easier to synthesize probability distributions from random numbers. Each one is a minor change if good algorithms can be found.

- Add floating point numbers. This would make some calculations easier since the user wouldn't have to worry about scaling. It would be a

major change, however, since it affects the compiler and many places in the interpreter. There is also some question whether a reasonable floating point package could be implemented in the 1.5K or so free space left in the ROM.

- Allow user programs to read the value of the last exception from a special variable. This would allow an exception handler that could report the exception that occurred regardless of what it was. This is of minor to average difficulty.

- Report exceptions causing processes to terminate to their parent. Currently the parent can't tell whether its child finished normally or due to an exception. Minor to average complexity.

- Allow multiple calls with a single call message. For example:

  A -> B -> C

  If A,B and C are processes and A calls B then B should be able to call C with the same call message. When C returns the call message it should go to B, and when B returns it, it should go back to A. This would allow a very natural progression of data down through the various layers of a protocol and back up. Currently only one level of calling is permitted, although the called process can FORWARD the call message to another process, but then it is out of the loop and a return sends it back to the original caller.

  There may be some tricky implications here, such as what happens if an intermediate process gets cancelled while it is waiting and others are waiting on it? This is an average change, and a very worthwhile one.

- Call messages and messages should be combined into a single data type. Messages are currently pretty useless since call messages can already be sent or called, while messages can only be sent. All of the operations on messages should be retained, but there is no reason to have two separate data types for essentially the same thing. This is probably only a minor change, just affecting the compiler, since

the interpreter already treats the two as the same. The reason that this wasn't implemented earlier was that it came from the original language and we wanted to retain as much of it as possible.

- There should be a third basic operation on messages besides send and call. This would be asynchronous like a send (i.e. the sender would not halt), but the data would eventually be returned like a call. We could implement this in one of two ways. The message would just be empty for a while (which could be tested with the EMPTY function) and then it would magically reappear, or we could specify a queue that it be returned to when the operation is performed.

  It often happens that a process needs to call another function, but it can still profitably do other things while the call is being processed. This is not that important on a single processor machine, but it could improve performance greatly on a multiprocessor such as (CLUMPS). One solution might be just to split up a process into two processes, but often this isn't possible if both parts need the same data. This would probably be a major change since the semantics need to be investigated, but it should be worth doing.

- Optimize expressions with a single constant. Things like adding zero, multiplying by a power of two, or comparing to zero are currently not optimized by the compiler the way they could be. This only affects one routine in the compiler and would be a minor change.

- Combine queues and pointers to queues into a single object type, allowing any process pointing to a queue to remove messages from it. This is a simple but potentially dangerous change. It mostly needs some thinking, but the implementation would be minor.

- Allow objects other than queues to be shared between processes. In particular, tables should be shareable. Again a major change, but probably very useful. This would need a lot more thought.

- Add constants to the NIL compiler such as in Pascal. This is an average change.

7.2.3 Operating System

- Have the operating system run a clock synchronization algorithm when it first starts up. This would be fairly simple to do by having each node ask its parent for the time a few times, subtracting the estimated the transaction response time, and averaging the results. This would be an average modification.

- As discussed in the previous chapter, set up the monitor program to control the switch. Generating the random topologies could be complicated, but just interfacing with the switch is minor.

Appendix A

Hardware Reference Manual

A.1 Introduction

This appendix defines (CLUMPS) hardware-software interface. It defines the hardware's structure (but not implementation) and how it will behave. Since (CLUMPS) is a memory mapped machine, most of this document describes the special functions available to its processors at certain memory locations.

A.2 Structure

(CLUMPS) is a multiprocessor machine. It has three basic parts: processing units, global bus, and shared resources. A processing unit consists of a microprocessor, memory, a serial interface, a timer, and an interface to the global bus. All processing units (also called branches) are identically constructed. A machine can have up to eight branches.

The global bus connects the individual branches to the shared resources. Since only one processor can connect to the shared resources at a time, the bus contains an arbiter to serialize the use of the global bus. All processing units have equal priority, the arbiter ensuring that no branch gets two bus accesses while another is waiting.

The shared resources (sometimes called global memory) consist of a large memory and a number of memory mapped special functions, which are detailed in a later section.

A.3 Memory Map

Each of the processors has a 64K address space which is divided among individual and shared (local and global) resources. The local resources are the branch's memory, serial port and timer. These are accessible ONLY by that branch's processor, and NOT via the global bus. The shared resources are accessible to every processor and the host.

Major breakdowns of the memory map are:

Addresses (hex)	Description
0000-007F	local functions
0080-009F	global functions
00A0-D7FF	global memory
D800-FFFF	local memory

A.4 Processing Unit (Branch)

A branch contains the following major parts: 68A09 microprocessor, 68B50 asynchronous serial interface, 8254 counter/timer, 68764 8Kx8 EPROM, 6116 2Kx8 static RAM, and global bus interface. They are all connected together with a standard 6809 bus: 16 bit address bus, 8 bit bidirectional data bus, R/(W) line, E and Q clocks. The 68A09 is run with a 6 MHZ clock crystal, so that the E and Q clocks run at 1.5 MHZ. The 8254 acts as programmable square wave generator to provide different baud rates for the 68B50 and interrupts for the 68A09.

The 6809 has three interrupt lines: FIRQ, IRQ, and NMI. FIRQ is attached to the 6850's interrupt output. IRQ is triggered by an 8254 counter reaching zero. The rate of counting is determined by the current

serial port baud rate. NMI is common to all the branches, and is generated by the global interrupt control (see section 5).

Since the processor has the ability to quickly address the lowest 256 bytes in the memory map, we have placed the peripherals there. 112 bytes of local RAM is also mapped into this section for fast access.

The detailed local memory map:

Addresses (hex)	Description
0000-006F	local RAM (aliased with D800-D86F)
0070-0076 even	6850 control/status register
0071-0077 odd	6850 data register
0078,0079	8254 counter 0
007A,007B	8254 counter 1
007C,007D	8254 counter 2
007E,007F	8254 control word
D800-DFFF	local RAM
E000-FFFF	local EPROM

A.5 Global Interface

The interface between global memory and the branches is as simple as possible. The common (tri-state) lines are a 16 bit address bus and an 8 bit data bus. The R/(W) is a common open collector line.

Each branch also has its own (REQ) and (ACK) lines. When a branch wants a global memory cycle it pulls its (REQ) line low. When the global arbiter responds by sending the (ACK) low, the branch puts its address on the address bus, releases the (REQ), and if it wants a write cycle it will pull the R/(W) line low and put its data on the data bus. When the (ACK) goes high the branch disconnects itself from the global buses, and in the case of a read, it latches the data from the data bus.

A.6 Global Functions

There are several special shared resources in addition to the memory.

There are two constant locations. One reads the machine's id number (0-255) which is set by a DIP switch. The other gives the branch's id number (0-7). Neither of these locations can be written.

A system reset occurs during power on, and when the reset location is written. An NMI occurs when the NMI location is written. Note that the reset and NMI locations overlap the global RAM. This is intentional; the processor doing the reset or NMI can write its id number (or any other byte) into the corresponding memory location concurrently with the reset or NMI operation. These locations can subsequently be read to determine, for example, the id number of the branch that executed the reset or NMI.

An 8254 timer is also located in the global area for use as a real time clock.

The detailed global memory map:

Addresses (hex)	Description
0000-007F	global RAM
0080-0086 even	Node ID (read only)
0081-0087 odd	Branch ID (read only)
0088,0089	8254 counter 0
008A,008B	8254 counter 1
008C,008D	8254 counter 2
008E,008F	8254 control word
0090-0097	NMI on write
0098-009F	RESET on write
0090-FFFF	global RAM

Appendix B

Software Reference Manual

B.1 NIL Syntax

The following is a syntax description of our modified version of NIL. It is in a form compatible with our parser generator. Capitalized words, punctuation and capital letters preceded by # are terminals. Lower case words preceded by # are actions (compiler functions that will be called). The right hand side of all statements starts with a terminal (or null) so no lookahead is necessary during parsing.

```
comp_unit       #= #I #procname : PROCESS proc_head semi #set_type type_defs
                   #set_declare proc_defn END proc_tail semi #cleanup

proc_head       #= ( #I #firstid : #I #set_np #iddef )
proc_head       #= #emptycm

semi            #= ;
semi            #=

type_defs       #= #I #firstid IS type_spec semi type_defs
type_defs       #=

type_spec       #= BOOL string_spec #booldef
type_spec       #= BOOLEAN string_spec #booldef
type_spec       #= CALL MESSAGE #cmessdef ( declares ) return_spec #endcmessdef
type_spec       #= CHAR string_spec #chardef
type_spec       #= CHARACTER string_spec #chardef
type_spec       #= COMPONENT #compdef
type_spec       #= INT string_spec #intdef
type_spec       #= INTEGER string_spec #intdef
type_spec       #= LONG string_spec #longdef
type_spec       #= MESSAGE #messdef ( declares ) #endfielddef
type_spec       #= TABLESET #tsdef ( tsdeclares ) #endtsdef

type_desc       #= #I type_desc1 #iddef
type_desc       #= type_spec
```

```
type_desc1      #= ACCEPTPORT #set_ap
type_desc1      #= CALLPORT #set_cp
type_desc1      #= RECEIVEPORT #set_rp
type_desc1      #= SENDPORT #set_sp
type_desc1      #= #set_np

declares        #= #I #firstid ids : type_desc semi declares
declares        #=

ids             #= , #I #uniqueid ids
ids             #=

return_spec     #= RETURNS ( #I #firstid : type_desc ) #set_return
return_spec     #= #set_noreturn

tsdeclares      #= #I #firstid tablerow0 semi tsdeclares
tsdeclares      #=

tablerow0       #= : tablerow
tablerow0       #= ids : row_def

tablerow        #= TABLE key_spec #tabledef ( tdeclares ) #endtabledef
tablerow        #= row_def

row_def         #= ROW IN #I #rowdef

key_spec        #= KEY ( #I ) #set_key
key_spec        #= #set_nokey

tdeclares       #= #I #firstid ids : tdeclare semi tdeclares
tdeclares       #=

tdeclare        #= ROW IN #I #rowdef
tdeclare        #= type_desc

string_spec     #= STRING #set_string
string_spec     #= #set_nostring

proc_defn       #= DECLARE declares #set_create proc_defn1
proc_defn       #= CREATE PHASE #set_create block #set_main

proc_defn1      #= CREATE PHASE block #set_main proc_defn2
proc_defn1      #= MAIN PHASE #set_main block

proc_defn2      #= MAIN PHASE block
proc_defn2      #=

proc_tail       #= PROCESS
proc_tail       #= #I

id              #= #I #identify

identifier      #= id func
```

```
block           #= #temppush statements #temppop

statements      #= ; statements
statements      #= #I statements1 nextstmt
statements      #= ACCEPT id FROM identifier #receive nextstmt
statements      #= ALLOCATE id key_stmt find_order #allocate nextstmt
statements      #= CALL id ( #procbeg proc_list_end #callend nextstmt
statements      #= COMCALL identifier #push0 #communicate nextstmt
statements      #= COMSEND identifier #push1 #communicate nextstmt
statements      #= COMPLETE identifier #wait nextstmt
statements      #= CONNECT id TO expression #connect nextstmt
statements      #= CREATE id ( identifier )
                     MODULE ( expression ) #create nextstmt
statements      #= DELETE id #delete nextstmt
statements      #= DISCARD identifier #discard nextstmt
statements      #= FIND id find_spec #findstmt nextstmt
statements      #= FORWARD identifier TO identifier #send nextstmt
statements      #= LEAVE leave_stmt #leave nextstmt
statements      #= LOSE id #discard nextstmt
statements      #= MOVE identifier TO id #movestmt nextstmt
statements      #= PUBLISH id AS expression #publish nextstmt
statements      #= RECEIVE id FROM identifier #receive nextstmt
statements      #= REMAP identifier AS id #remap nextstmt
statements      #= RETURN identifier return_tail #returnstmt nextstmt
statements      #= SEND identifier TO identifier #send nextstmt
statements      #= SLEEP #sleep nextstmt
statements      #= TIME = expression #settime nextstmt
statements      #= UNPUBLISH id #unpublish nextstmt
statements      #= WAIT id #wait nextstmt
statements      #= ZEROSTRING ( id #chkstr , expression #chkint )
                     #zerostring nextstmt
statements      #= IF #pushif #pushlabel expression #tempclear #ifthen
                   THEN block else END #poplabel IF nextstmt
statements      #= DO #pushdo #pushlabel #dodo block ons
                     END #doend #poplabel DO nextstmt
statements      #= SELECT #pushselect #pushlabel select_body
                     END #poplabel SELECT nextstmt
statements      #= REPEAT #pushrepeat #pushlabel repeat_stmt
                     END #poplabel REPEAT nextstmt
statements      #=

statements1     #= : #pushlabel cntrl_stmt #poplabel
statements1     #= [ #identify #chkstr expression #indxsetup index_assign ]
                   = expression #subassign
statements1     #= = #identify expression #assign

index_assign    #= ; index_limit #substring
index_assign    #= #indx

nextstmt        #= #tempclear statements

find_spec       #= WHERE #findwhere ( expression ) #chkbool
find_spec       #= KEY ( expression ) #push1
```

```
find_spec       #= find_order #push0

find_order      #= FIRST #push1
find_order      #= LAST #push2
find_order      #= AFTER id #push3
find_order      #= BEFORE id #push4
find_order      #= #push0

key_stmt        #= KEY ( expression )
key_stmt        #= #push0

leave_stmt      #= DO #pushdo
leave_stmt      #= EXCEPTION ( except )
leave_stmt      #= IF #pushif
leave_stmt      #= REPEAT #pushrepeat
leave_stmt      #= SELECT #pushselect
leave_stmt      #= #I

return_tail     #= EXCEPTION ( except )
return_tail     #= #push0

cntrl_stmt      #= IF #pushif #pushlabel expression #tempclear #ifthen
                     THEN block else END #poplabel if_tail
cntrl_stmt      #= DO #pushdo #pushlabel #dodo block ons
                     END #doend #poplabel do_tail
cntrl_stmt      #= SELECT #pushselect #pushlabel select_body
                     END #poplabel select_tail
cntrl_stmt      #= REPEAT #pushrepeat #pushlabel repeat_stmt
                     END #poplabel repeat_tail

else            #= ELSE #pushif #leave #ifend block
else            #= #ifend

if_tail         #= IF
if_tail         #= #I #cmplabel

do_tail         #= DO
do_tail         #= #I #cmplabel

select_tail     #= SELECT
select_tail     #= #I #cmplabel

repeat_tail     #= REPEAT
repeat_tail     #= #I #cmplabel

ons             #= ON #doon ( except excepts ) block ons
ons             #=

excepts         #= , #doexcept #donotlastexcept except excepts
excepts         #= #doexcept #dolastexcept

except          #= ERROR #push1
except          #= ALIASING #push2
except          #= BOUNDARY #push3
```

```
except          #= DEPLETION #push4
except          #= DISCONNECTED #push5
except          #= DUPLICATE #push6
except          #= FORMAT #push7
except          #= NOT FOUND #push8
except          #= OVERFLOW #push9
except          #= SELECT FAIL #push10
except          #= SIZE #push11
except          #= TYPE MISMATCH #push12
except          #= PARAMETER #push13
except          #= #N #fixexcept

select_body     #= WAIT #repeat event_whens when_otherwise #repeatend
select_body     #= ( expression ) #select value_whens otherwise
select_body     #= #true #select value_whens otherwise

value_whens     #= WHEN ( expression when_exprs ) block
                     #pushselect #leave #whenend value_whens
value_whens     #=

when_exprs      #= , #selectexpr expression when_exprs
when_exprs      #= #selectlastexpr

when_otherwise  #= OTHERWISE block #pushselect #leave
when_otherwise  #= #sleep

otherwise       #= OTHERWISE #tempclear #otherwise block
otherwise       #= #otherwise #push10 #leave

event_whens     #= WHEN ( id ) #selectevent #ifthen block
                     #pushselect #leave #ifend event_whens
event_whens     #=

repeat_stmt     #= WHILE #repeat ( expression ) #while1 block
                     #repeatend #outstack
repeat_stmt     #= UNTIL #until1 ( expression ) #until2 block
                     #repeatend #outstack
repeat_stmt     #= FOR for_iterator
repeat_stmt     #= #repeat block #repeatend

for_iterator    #= ( id = expression up_down expression )
                     #for1 block #outstack
for_iterator    #= id #forrow1 #doon #push8 #doexcept
                      #dolastexcept #pushrepeat #leave #doend #poplabel
                      where #forrow2 block #repeatend

where           #= WHERE ( expression ) #chkbool #tempclear
where           #= #true

up_down         #= TO #push0
up_down         #= DOWNTO #push1

index_expr      #= [ #chkstr expression #indxsetup index_end ]
index_expr      #=
```

```
index_end       #= ; index_limit #substring
index_end       #= #indx #indxref

index_limit     #= * #endlimit
index_limit     #= expression #indxsetup

term            #= #C
term            #= #N
term            #= #L
term            #= #S index_expr
term            #= TRUE  #true
term            #= FALSE #false
term            #= ( expression ) index_expr
term            #= LENGTH  ( expression ) #length
term            #= ABS     ( expression #chkint ) #nabs
term            #= ORD     ( expression ) #ord
term            #= CHR     ( expression ) #chr
term            #= INTEGER ( expression ) #convint
term            #= LONG    ( expression ) #convlong
term            #= COMPLETED ( identifier ) #completed
term            #= EMPTY ( identifier ) #empty
term            #= COMSTATE ( expression ) #comstate
term            #= TIME #ntime
term            #= NODEID #nodeid
term            #= BRANCHES #branches
term            #= PROCESSORS #processors
term            #= RANDOM #random
term            #= RANDOML #randoml
term            #= BAUDRATE ( expression #baud )
term            #= +   term #chkint
term            #= -   term #chkint  #neg
term            #= ~   term #chkint  #notint
term            #= NOT term #chkbool #notbool
term            #= identifier index_expr

op_multiplicative #= *   #chkint term #chkint #mul op_multiplicative
op_multiplicative #= /   #chkint term #chkint #div op_multiplicative
op_multiplicative #= MOD #chkint term #chkint #mod op_multiplicative
op_multiplicative #= &   #chkint term #chkint #and op_multiplicative
op_multiplicative #= SHL #chkint term #chkint #shl op_multiplicative
op_multiplicative #= SHR #chkint term #chkint #shr op_multiplicative
op_multiplicative #=

multiplicative  #= term op_multiplicative

op_additive     #= + #chkint multiplicative #chkint #add op_additive
op_additive     #= - #chkint multiplicative #chkint #sub op_additive
op_additive     #= | #chkint multiplicative #chkint #or  op_additive
op_additive     #= ^ #chkint multiplicative #chkint #xor op_additive
op_additive     #=

additive        #= multiplicative op_additive
```

```
op_concatenation #= ! additive #tempclear #concatenate op_concatenation
op_concatenation #=

concatenation    #= #temppush additive op_concatenation #swap #tempmerge

op_relational    #= < op_less
op_relational    #= > op_great
op_relational    #= = concatenation #equal op_relational
op_relational    #=

op_less          #= = concatenation #lessequal  op_relational
op_less          #= > concatenation #notequal   op_relational
op_less          #=   concatenation #less       op_relational

op_great         #= = concatenation #greatequal op_relational
op_great         #=   concatenation #great      op_relational

relational       #= concatenation op_relational

op_expression    #= AND #chkbool and_clause op_expression
op_expression    #= OR  #chkbool or_clause  op_expression
op_expression    #= XOR #chkbool relational #chkbool #xor op_expression
op_expression    #=

and_clause       #= THEN #tempclear #andthen1 relational #chkbool #tempclear
                       #andthen2
and_clause       #= relational #chkbool #and

or_clause        #= ELSE #tempclear #orelse1 relational #chkbool #tempclear
                       #orelse2
or_clause        #= relational #chkbool #or

expression       #= #temppush relational op_expression #swap #tempmerge

func             #= ( #funcbeg proc_list_end #callend
func             #=

proc_list_end    #= ) #noparm
proc_list_end    #= expression procone

procone          #= ) #oneparm
procone          #= #firstparm procs

procs            #= , expression #procparm procs
procs            #= )

expression_list  #= ( #mark expression_list_end

expression_list_end #= )
expression_list_end #= expression expressions

expressions      #= , expression expressions
expressions      #= )
```

B.2 Intermediate Language Syntax

The following is a syntax description of the Intermediate Language source code used by the assembler. Like the NIL syntax in the previous section it is compatible with our parser generator.

```
asm_unit        #= first_pass second_pass

first_pass      #= statements END

second_pass     #= statements END

term            #= #I
term            #= #C
term            #= #N
term            #= ( expression )
term            #= + term
term            #= - term
term            #= ~ term
term            #= .

statements      #= #I id
statements      #= #S seperate statements
statements      #= #C op_statement
statements      #= #N op_statement
statements      #= ( expression ) op_statement
statements      #= + term op_statement
statements      #= - term op_statement
statements      #= ~ term op_statement
statements      #= . op_statement
statements      #=

id              #= : seperate statements
id              #= EQU expression seperate statements
id              #= op_statement

op_statement    #= op_multiplicative op_additive op_shift
                   generate seperate statements

generate        #= @
generate        #=

seperate        #= ;
seperate        #= ,
seperate        #=

op_multiplicative #= *    term op_multiplicative
```

```
op_multiplicative #= /    term op_multiplicative
op_multiplicative #= MOD  term op_multiplicative
op_multiplicative #= &    term op_multiplicative
op_multiplicative #=

multiplicative  #= term op_multiplicative

op_additive     #= +    multiplicative op_additive
op_additive     #= -    multiplicative op_additive
op_additive     #= |    multiplicative op_additive
op_additive     #= XOR  multiplicative op_additive
op_additive     #=

additive        #= multiplicative op_additive

op_shift        #= < < additive op_shift
op_shift        #= > > additive op_shift
op_shift        #=

expression      #= additive op_shift
```

B.3 Intermediate Language Opcodes

The following is a list of the intermediate language instructions recognized by the assembler, and descriptions of their operation.

LEGEND:

Capitalized symbols represent values on the stack. Lower case symbols represent parameters that follow the opcodes in the intermediate language code. Most of these parameters are objects which are represented by lists of single byte offsets (see text).

->	Contents of the stack before and after execution of the operation. By convention the top of the stack is always on the right in any list of stack elements.
[]	Optional parameter.
A	Address. 16 bits. Can be relative or absolute depending on context.
I	Integer. Has a value from -2**15 to 2**15-1.
L	Long. Value between -2**31 and 2**31-1.
S	String description. Consists of address and length. S1 is the same as A1 I1.
TF	Boolean true (1) or false (0).
V	Value. Depends on context.
acc	Access mode of a row (single byte constant). They are: 0-nothing, 1-detached, 2-update, 3-read, 4-refer, 5-marked.

```
add     16 bit relative address constant.
byte    8 bit constant.
comp    Component.
key     Object used as key.
long    32 bit constant.
mes     Message.
obj     Object (type based on context).
pri     Primary.
row     Row.
sec     Secondary.
str     String.
word    16 bit constant.
```

Stack	Op	Params	Description
Constant			
-> -4	ld_4		Load minus four on stack (integer)
-> -3	ld_3		Load minus three
-> -2	ld_2		Load minus two
-> -1	ld_1		Load minus one
-> 0	ld0		Load zero
-> 1	ld1		Load one
-> 2	ld2		Load two
-> 3	ld3		Load three
-> 4	ld4		Load four
-> I1	ldcb	byte	Load byte constant (unsigned)
-> I1	ldcw	word	Load word constant
-> S1	ldcs	add	Load string address and length given its relative address
Stack Manipulation			
I1 ->	drop		Get rid of top of stack
L1 ->	dropl		Get rid of top two stack elements
I1 -> I1 I1	dup		Duplicate top of stack
L1 -> L1 L1	dupl		Duplicate long on top of stack
I1 I2 -> I2 I1	swap		Exchange top two stack elements
L1 L2 -> L2 L1	swapl		Exchange top two longs
I1 L1 I2 -> I2 L1 I1	swap4		Exchange first and fourth stack elements
I1 I2 -> I1 I2 I1	over		Duplicate element next to top
I1 I2 I3 -> I2 I3 I1	rot		Rotate top three elements
I1 I2 I3 -> I3 I1 I2	unrot		Undo rotate of top three elements
Arithmetic			
I1 I2 -> I3	add		I1+I2
I1 I2 -> I3	sub		I1-I2
I1 I2 -> L1	mul		I1*I2 (with long result)
I1 I2 -> I3	div		I1/I2
I1 I2 -> I3	modulo		I1 MOD I2
L1 L2 -> L3	addl		L1+L2
L1 L2 -> L3	subl		L1-L2
L1 L2 -> L3	mull		L1*L2
L1 L2 -> L3	divl		L1/L2
L1 L2 -> L3	modulol		L1 MOD L2

```
   I1 -> I2     abs            Absolute value
   L1 -> L2     absl           Absolute value long
   I1 -> I2     neg            Change sign
   L1 -> L2     negl           Change sign long
   I1 -> L1     il             Convert integer to long
   L1 -> I1     li             Convert long to integer
I1 I2 -> I3     and            Bitwise logical and
I1 I2 -> I3     or             Bitwise logical or
I1 I2 -> I3     exor           Bitwise exclusive or
   I1 -> I2     not            Bitwise logical not
   L1 -> L2     notl           Bitwise logical not long
I1 I2 -> TF     lt             I1 less than I2
L1 L2 -> TF     ltl            L1 less than L2
I1 I2 -> TF     eq             I1 equal to I2
L1 L2 -> TF     eql            L1 equal to L2
   I1 -> TF     eq0            I1 equal 0 (boolean not)
I1 I2 -> I3     lsl            Logical shift I1 left I2 times
L1 I1 -> L2     lsll           Logical shift left long
I1 I2 -> I3     lsr            Logical shift right
L1 I1 -> L2     lsrl           Logical shift right long
      -> I1     rand           Random integer
      -> L1     randl          Random long
      -> L1     time           time of day
   L1 ->        settime        set the time of day
```

Memory

```
   A1 -> I1     ldb            Load byte at address A1 (unsigned)
I1 A1 ->        stb            Store byte I1 at address A1
   A1 -> I1     ldw            Load word at address A1
I1 A1 ->        stw            Store word I1 at address A1
   A1 -> L1     ldl            Load long at address A1
L1 A1 ->        stl            Store long L1 at address A1
   A1 -> A2     relabs         Change relative address to absolute
   I1 -> A1     allblk         Allocate block of length I1
   A1 ->        relblk         Discard block at A1
   A1 -> I1     blklen         Length of given block
   A1 ->        lock           Lock the semaphore at A1 (dangerous)
   A1 ->        unlock         Unlock the semaphore at A1
```

General Object

```
      -> V1     ldo  obj       Load object
   V1 ->        sto  obj       Store object
      -> A1     lda  obj       Load pointer to data
      -> TF     emp  obj       Test if object is initialized
                disc obj       Return storage for the object
      -> I1     length obj     Load length of str, table, primary or sec
                move obj1 obj2 Transfer data: Obj1=source, Obj2=dest
                remap obj1 obj2 Create obj2 from obj1's data
```

String

```
S1 S2 ->        scopy          Copy data from string 1 to string 2
                                 Error if lengths are not the same
S1 S2 ->        sconcat str    Concat string 1 and string 2 to form str
S1 I1 -> A1     sindex         Return address of the I1st byte of S1;
```

```
                                        Error if index is out of range
S1 I1 I2 -> S2   substr                 Calculate substring from string, low
                                          index(I1) and length(I2); Error if
                                          out of bounds
S1 S2 -> TF      seq                    Compare two strings for equality
S1 S2 -> TF      sltb                   Compare two byte strings, S1 < S2
S1 S2 -> TF      sltw                   Compare two word strings, S1 < S2
S1 S2 -> TF      sltl                   Compare two long strings, S1 < S2
   I1 ->         zerostr str            Create string of zeros of length I1
```

Primary, Secondary, Message and Component

```
   S1 ->         pub    pri str         Put primary in catalog with name S1
                 unpub  pri             Take primary out of catalog
                 con1   str sec         Look up str in catalog, attach sec
                 con2   pri sec         Use given primary to make connection
                 send   mes sec         Add message to queue found by secondary
                 call   mes sec         Add message to queue found by secondary,
                                          then wait for return
                 rec    pri mes         Take message from queue, or error if empty
                 ret    mes byte        Return message to caller with given error
   S1 ->         create mes             Calls Op Sys to start a new process
                                         named S1, called with mes
      -> TF      complete comp          Check if component object has completed,
                                          putting true or false on stack
                 comsend mes            Send commess or cominitmess to port
                 comcall mes            Call port with commess or cominitmess
```

Table and Row

```
                 alc   row/mess [key]   Allocate record or message
                 alca  row1 [key] row2  Allocate record after row2
                                          after nothing means first
                 alcb  row1 [key] row2  Allocate record before row2
                                          before nothing means last
                 fnda  row1 row2        Find record after row2
                                          after nothing means first
                 fndb  row1 row2        Find record before row2
                                          before nothing means last
                 fndk  row  key         Find record with key
                 delete  row            Detach record and discard its data
```

Flow of Control

```
   TF ->         bz      add      Branch if top of stack is zero
   TF ->         bnz     add      Branch if top of stack not zero
                 bra     add      Branch unconditional
                 bsr     add      Branch to subroutine
                 rts              Return from subroutine
                 signal  byte     Effect the exception as if it occurred
                 wait    pri      Wait until primary receives message
                 wait    comp     Wait until components finishes
                 sleep            Swap process out to ready queue
                 stop             Mark process as completed
```

Miscellaneous

```
                 debug            Not implemented
```

nop	No operation

Appendix C

Operating Manual

This appendix gives operating instructions for all of the host based system software: global type database, compiler, assembler, and monitor.

C.1 Global Database

The program "global" manipulates a database of NIL data types that will be shared among your processes. It expects to find the data file "global.sym" on your default disk. Before running global, make up a file with the new data types you want to install. Then just type "global". The commands it accepts are as follows:

A	Display the contents of the database.
H	Help information.
L	Display all of the data type names currently in the database.
P	Directs future display output to the printer.
Q	Quit.
R	Display the definition of a given data type.
S	Directs future display output to the screen.
W	Write the contents of a file into the database. This is the only command that actually changes the database. All the others just report what is in it. If the file you write contains a data type definition that is already in the database then the old one will be replaced by the new one. Other than by deleting the file "global.sym" this is the only way of getting rid of

an old definition.

C.2 Compiler

The compiler expects your NIL source code in a file with the extension ".NIL". It also expects the file "global.sym" on the default disk drive. This file must contain any shared data types used in the program currently being compiled. The source code must contain only one process. The compiler is run by entering:

compiler filename

It creates two files. One called "filename.LST" is a copy of the source code together with any warning messages, error messages, and shared data type definitions at the places that they occurred. The other file has as its filename the name of the process in the NIL source code, and it uses the extension ".INT". This file contains the intermediate code result of the compilation. It can be printed out if one is curious as to how the compiler works.

C.3 Assembler

The assembler takes the output of the compiler and converts the process into the coded form that can be run by the (CLUMPS) interpreter. It is run just like the compiler. It expects a file called "assemble.def" on the default drive. This file contains definitions of all the opcode bytes (it is automatically included during assembly). The assembler's input file should have the extension ".INT". To run it just enter:

assemble filename

It also creates two files: a listing file with the errors and the included definitions, and the object code file with the extension ".COB".

No intermediate code files that the assembler produces should have any assembly errors.

C.4 Monitor

The monitor program is used to control (CLUMPS) from the host. It sends the operating system, user processes and other files to (CLUMPS), and records and/or displays simulation results from (CLUMPS). Before running the monitor (CLUMPS) must be powered on and both it and the switch must be reset. The procedure is as follows:

1. Turn power on to (CLUMPS), the PC (host), and the terminal used to control the switch.

2. Initialize the switch with a command of the form:

   INIT 8000 10

   Where 8000 is the address of the switch memory (in hex), and 10 is the number of lines in the switch (in hex).

3. Reset all the (CLUMPS) nodes with their hardware reset buttons.

4. Program the switch for the connections desired. The commands are in the form:

   C node-in node-out

   Where C is the connect command (use D for disconnect), node-in is the number of the communication line into the switch (in hex), and node-out is the number of the communication line going out of the switch (in hex). Remember that for bidirectional connections both directions must be programmed separately.

5. Run the monitor program by entering "monitor".

The monitor expects all of the object code files it will need to be on the default disk drive. At a minimum this means the operating system file "os.cob", and the main user program "user.cob". To run the monitor just enter "monitor". All monitor commands are control characters. This prevents messing up a simulation run by accidentally pressing a key. The commands are as follows:

^A — Toggle raw mode. When in raw mode all received messages are treated like log messages whether or not they have a leading zero.

^C — Quit. Asks if you want to leave the monitor. The monitor can be stopped and resumed while (CLUMPS) is reset without affecting the system.

^K — Compose a message at the keyboard to send to (CLUMPS). First enter a string of message data. To include a non-ascii character in the string enter a $ followed by two hexadecimal digits. These three characters will be turned into a single byte. Next enter the node id of the destination in hex. This is turned into a single byte which is concatenated onto the front of the data. The whole message is then sent.

^L — Toggle logging. When logging is on all log messages received from (CLUMPS) are stored to disk. They are placed in a file called "clumps.log". If such a file exists when the monitor is invoked, then it is renamed to "clumps.bak", and a new log file is created.

^O — Send the operating system to (CLUMPS). This is the usual first operation after the monitor is started.

^Q — Display the most recently received 8 messages. This is mostly useful for debugging.

^R — Reset (CLUMPS). Sends a message that causes all nodes to go into their power on reset state. This is used to end one simulation and start another.

^S — Freeze processing until another key is pressed. This is used to halt the screen if messages are coming in to quickly to read.

^W — Send a file to (CLUMPS). This is like ^K, but the data comes from a file instead of the keyboard.

^X — Toggle display mode. When display mode is on all received messages will be displayed on the screen. They will be truncated to reduce the screen clutter if they are greater than 40 characters. The complete message is

always logged, however.

Appendix D

Sample NIL program

The program that follows illustrates a complete packet radio simulation system (see section 3.2 for a description). The operating system NIL program is included for completeness. The system consists of eight different processes representing the various layers of protocol.

D.1 Shared Data Types

The following are data types common to all the processes.

```
arqstart IS CALL MESSAGE (
    id: CHAR;
    port: ethermess CALLPORT;
    ) RETURNS (cmp: COMPONENT);
charmess is CALL MESSAGE(c:char) RETURNS(cmp:COMPONENT);
cominitmess IS CALL MESSAGE
     ( flag:   CHAR;
       branch: CHAR; { 0 to BRANCHES-1 }
       baud:   INTEGER; { if zero, not changed }
       errate: INTEGER; { errors per 2**24 bytes }
       delay:  INTEGER; { in clock ticks }
       carriersense: INTEGER; { neg = radio, 0 = wire, pos = car sense delay
                                in 16ths of a bit time }
       recp: commess CALLPORT;
       arcp: commess CALLPORT; );
commess IS CALL MESSAGE
     ( flag:      CHAR;
       branch:    CHAR;
       timestamp: LONG;
       data:      CHAR STRING; );
createmess IS CALL MESSAGE() RETURNS(name:CHAR STRING);
ethermess IS CALL MESSAGE ( data: CHAR STRING );
logmess  IS MESSAGE(data:CHAR STRING);
procmess IS MESSAGE
     ( pcount: INTEGER; {program counter}
```

```
        pcode : INTEGER; {tuple containing code}
        pdata : INTEGER; {top-level data}
        parent: INTEGER; {owner who could be waiting on this process}
        psem  : CHAR;    {semaphore}
        status: CHAR;    {state of process} );
   readmess IS CALL MESSAGE(data,result:CHAR STRING);
   schedulemess is CALL MESSAGE(limit:LONG);
```

D.2 Operating System

```
{5/21/86 DRH}
{Main operating system nil process}
{Does log, read, createproc, discardproc and handles all os branches}
os: PROCESS
DECLARE
   user: COMPONENT; { pointer to the main user program }
   lp: logmess RECEIVEPORT;
   lm: logmess;
   rp: readmess ACCEPTPORT;
   rm: readmess;
   cm: commess;
   parentp,childp: commess ACCEPTPORT;
   pointer,i: INTEGER;
   routes: CHAR STRING;
   cim: cominitmess;
   crp: createmess ACCEPTPORT; { creque }
   crm: createmess;
   dp: procmess RECEIVEPORT; { disque }
   pm: procmess;
   code: TABLESET
      ( t: TABLE KEY(name) (name,data: CHAR STRING);
        r: ROW IN t; );
   ltemp:long;

CREATE PHASE
   {SETUP THE GLOBAL RECEIVE BUFFER AREA}
   zerostring(routes,32); {maximum of 32 nodes}
   routes[0]=CHR(branches); {send to parent when going to node 0}

   {connect the os comm lines to parentp or childp}
   CONNECT cim.recp TO parentp;
   cim.branch=CHR(branches);
   COMSEND cim;

   {get user process from parent}
   cm.data=nodeid!"user.cob";
   cm.branch=CHR(branches);
   COMSEND cm;
```

```
    REPEAT FOR (i=branches+1 TO processors-1)
       {init the kids}
       CONNECT cim.recp TO childp;
       cim.branch=CHR(i);
       COMSEND cim;
       {send each one an operating system}
       ALLOCATE cm;
       {!
          ldcw $200@, {place where the os is}
          lda  V1_cm, V6_data,
          stw
       }
       cm.branch=CHR(i);
       COMSEND cm;
    END REPEAT;

    PUBLISH lp AS "log";
    PUBLISH rp AS "read";
    PUBLISH crp AS "createproc";
    PUBLISH dp AS "discardproc";

    { store address of crp, dp in fixed global locations }
    {!
    ldo V1_crp,
    ldcb $f0, {creque}
    stw
    ldo V1_dp,
    ldcb $f2, {disque}
    stw
    }

    {SET OUR CLOCK FROM PARENT?}

MAIN PHASE

    {Hopefully we have gotten a reply from our parent by now}
    WAIT parentp;
    RECEIVE cm FROM parentp;

    {check that this is the packet we wanted}
    IF cm.flag=CHR(0) AND THEN cm.data[0]=nodeid THEN

    {start the given process and go for it}
    ALLOCATE code.r KEY("user.cob");
    code.r.data=cm.data[1;*];
    DISCARD cm;
    ALLOCATE pm; { make a process for the user }
    {!
       lda V1_pm, 0,
       ldw              {store a pointer to this process in our component}
       lda V1_user,
       stw
       bsr STARTPROC@, {set pcount, pcode, put on process queue}
```

```
}

REPEAT
   DO
      SELECT WAIT
         WHEN (parentp)
            RECEIVE cm FROM parentp;
            IF cm.flag<>CHR(0) THEN
               {send the error message back to dad}
               cm.data=CHR(255)!nodeid!cm.flag;
               cm.branch=CHR(branches);
               COMSEND cm;
            ELSE IF cm.data[0]=CHR(255) THEN
               { parent is telling us to reset }
               LEAVE REPEAT; {quit the main loop}

               {convenient place to put the subroutines}
          {! MAKEPROC:  }
               DISCARD crm.name;
               ALLOCATE pm;
               {!
               {store the process in crm.name}
               lda V1_pm, 0,
               ldw
               lda V1_crm, V8_name,
               stw
               {allocate the data}
               ldo V1_code, V10_r, V11_data,
               drop
               dup
               ldw  {now have relative ptr to v1 descr}
               add  {now have absolute ptr to v1 descr}
               ld2
               add  {now have absolute ptr to length of v1}
               ldw  {now have length of v1}
               allblk
               sto V1_pm, V9_pdata,
               {move crm into the data as the initial message}
               {the data is in a regular message, not a call message}
               ld0  {clear out crm after moving it}
               lda V1_crm, 0,
               dup
               ldw  {now we have a ptr to crm}
               ldo V1_pm, V9_pdata,
               ld4,
               add  {now we have the data+4, i.e. the first variable}
               stw  {save crm in data+4}
               stw  {clear crm}
            STARTPROC:
               {set pcount to the start of the code}
               ldo V1_code, V10_r, V11_data,
               drop
               ld4,
               add
```

```
      sto V1_pm, V9_pcount,
      {set the pcode to the tuple}
      move V1_code, V10_r, 0, V1_pm, V9_pcode,
      ldcw $180@,
      sto  V1_pointer,
      send V1_pm, 0, V1_pointer,
      rts
      }
   ELSE IF cm.data[0]=nodeid THEN
      { this is a read or create reply to us }
      IF EMPTY(rm) THEN
         ALLOCATE code.r KEY(crm.name);
         code.r.data=cm.data[1;*];
         DISCARD cm;
         {setup pm from code.r}
         {! bsr MAKEPROC@ }
      ELSE
         rm.result=cm.data[1;*];
         DISCARD cm;
         RETURN rm;
      END IF
   ELSE { read reply to one of our children, forward it }
      cm.branch=routes[ORD(cm.data[0])];
      COMSEND cm;
   END IF END IF END IF
WHEN (childp)
   RECEIVE cm FROM childp;
   IF cm.flag<>CHR(0) THEN
      cm.data=CHR(255)!nodeid!cm.flag;
   END IF;
   IF cm.data[0]<>CHR(0) AND cm.data[0]<>CHR(255) THEN
      {Update our routing table}
      {HANDLE REQUESTS FOR TIME?}
      routes[ORD(cm.data[0])]=cm.branch;
      {We have a read request, first see if we have the data here}
      DO
         FIND code.r KEY(cm.data[1;*]);
         cm.data=cm.data[0]!code.r.data;
         DISCARD code.r;
      ON (not found)
         {if not, just pass the message along}
         cm.branch=CHR(branches);
      END DO
   ELSE {send to parent}
      cm.branch=CHR(branches);
   END IF;
   COMSEND cm;
WHEN (lp)
   RECEIVE lm FROM lp;
   cm.data=CHR(0)!lm.data;
   DISCARD lm;
   cm.branch=CHR(branches);
   COMSEND cm;
WHEN (dp)
```

```
            RECEIVE pm FROM dp;
            {set code.r from pm.pcode}
            {! move V1_pm, V9_pcode, V1_code, V10_r, 0, }
            DISCARD pm;
            DO
               DELETE code.r;
            ON (error)
               DISCARD code.r;
            END DO;
         WHEN (rp)
            IF EMPTY(rm) and EMPTY(crm) THEN {only do one at a time}
               RECEIVE rm FROM rp;
               cm.data=nodeid!rm.data; {ask parent for the data}
               cm.branch=CHR(branches);
               COMSEND cm;
            ELSE
               SLEEP;
            END IF
         WHEN (crp)
            IF EMPTY(rm) and EMPTY(crm) THEN {only do one at a time}
               RECEIVE crm FROM crp;
               DO
                  FIND code.r KEY(crm.name);
                  {setup pm from code.r}
                  {! bsr MAKEPROC@ }
               ON (not found)
                  {ask parent for the data}
                  cm.data=nodeid!crm.name;
                  cm.branch=CHR(branches);
                  COMSEND cm;
               END DO
            ELSE
               SLEEP;
            END IF
      END SELECT
   ON (error)
   END DO
END REPEAT
END IF

{stop the user}
DISCARD user;

{send reset message to the kids}
REPEAT FOR (i=branches+1 TO processors-1)
   cm.branch=CHR(i);
   cm.data=CHR(255);
   COMCALL cm;
END REPEAT

{invalidate all user branches}
REPEAT FOR (i=0 TO branches-1)
   {!
   ld1
```

```
      ldcw $101@, {valid is the byte after sem}
      ldo V1_i,
      ld4
      lsl
      add
      stb
      }
   END REPEAT;

   {wait 3 seconds}
   ltemp=time+30000;
   REPEAT WHILE (ltemp>time) SLEEP; END REPEAT;

   {reset the node}
   {!
   ld0
   ldcb $98,
   stb
   }

END PROCESS
```

D.3 Main User Process

```
{ simple radio protocol, demonstrates layering, routing and flow control }
user: PROCESS

DECLARE
   em: ethermess;
   ep: ethermess ACCEPTPORT;
   cc: charmess; {dummy for starting up the processes}
   c1,c2,c3,c4,c5: COMPONENT;
   lm: logmess;
   lp: logmess SENDPORT;
   schedule: schedulemess CALLPORT;
   router: ethermess CALLPORT;
   counts: INTEGER STRING;
   i,count: INTEGER;
   c: CHAR;

CREATE PHASE
   PUBLISH ep AS "application";
   CREATE c1(cc) MODULE("schedule.cob");
   CREATE c2(cc) MODULE("sender.cob");
   CREATE c3(cc) MODULE("arqdistr.cob");
   CREATE c4(cc) MODULE("router.cob");
   CREATE c5(cc) MODULE("receiver.cob");
   CONNECT lp TO "log";
```

```
   CONNECT router TO "router";
   CONNECT schedule TO "schedule";

MAIN PHASE
DO
   lm.data=nodeid!"starting";
   SEND lm TO lp;
   zerostring(counts,5);
   count=0;
   CALL schedule(20000); { wait for other to get started }

   REPEAT FOR (i=1 TO 3) { send initial messages }
      IF i<>ORD(nodeid) THEN
         counts[i]=1;
         zerostring(em.data,7);
         em.data[1]=nodeid;
         em.data[4]=CHR(i);
         em.data[5]=nodeid;
         em.data[6]=CHR(1);
         SEND em TO router;
      END IF;
   END REPEAT;

   REPEAT
      WAIT ep;
      RECEIVE em FROM ep; {receive a message}
      lm.data=em.data; {report it to the screen}
      SEND lm TO lp;
      i=ORD(em.data[5]);
      IF counts[i]<8 THEN {send another one}
         counts[i]=counts[i]+1;
         em.data[1]=nodeid;
         em.data[4]=CHR(i);
         em.data[5]=nodeid;
         em.data[6]=CHR(counts[i]);
         SEND em TO router;
      ELSE
         count=count+1; {finished sending to a node}
         IF count=2 THEN LEAVE REPEAT; {all done} END IF;
      END IF;
   END REPEAT;

   lm.data=nodeid!"ending";
   SEND lm TO lp;
   REPEAT CALL schedule(10000000); END REPEAT;
ON (error)
END DO;
lm.data=nodeid!"demo error";
SEND lm TO lp;

END PROCESS
```

D.4 Scheduler

```
{ process to help others wait }
{ accepts call messages with number of ticks to wait }
{ returns the call messages after the given amount of time passes }
schedule: PROCESS
DECLARE
   sm: schedulemess;
   sp: schedulemess ACCEPTPORT;
   ts: TABLESET (
       t: TABLE (sm:schedulemess);
       rfirst,r: ROW IN t; );

CREATE PHASE
   PUBLISH sp as "schedule";
   ALLOCATE ts.rfirst;
   ts.rfirst.sm.limit=time+36000000;

MAIN PHASE
   REPEAT
      IF ts.rfirst.sm.limit<=time THEN
      {return a call message if time}
         RETURN ts.rfirst.sm;
         DELETE ts.rfirst;
         FIND ts.rfirst FIRST;
      ELSE IF LENGTH(sp)<>0 THEN
      {add a new call message to our table}
         RECEIVE sm FROM sp;
         sm.limit=sm.limit+time;
         IF sm.limit<=ts.rfirst.sm.limit THEN
            ALLOCATE ts.rfirst BEFORE ts.rfirst;
            MOVE sm TO ts.rfirst.sm;
         ELSE
            REPEAT FOR ts.r
               IF ts.r.sm.limit>=sm.limit THEN LEAVE REPEAT; END IF;
            END REPEAT;
            ALLOCATE ts.r BEFORE ts.r;
            MOVE sm TO ts.r.sm;
            LOSE ts.r;
         END IF;
      ELSE SLEEP; {just wait if nothing to do}
      END IF; END IF;
   END REPEAT;

END PROCESS
```

D.5 Sender

```
{send messages until no carrier sense errors}
sender: PROCESS

DECLARE
   cm: commess;
   em: ethermess;
   ep: ethermess ACCEPTPORT;
   schedule: schedulemess CALLPORT;

CREATE PHASE
   PUBLISH ep AS "sender";
   CONNECT schedule TO "schedule";

MAIN PHASE
   REPEAT
      WAIT ep;
      RECEIVE em FROM ep;
      MOVE em.data TO cm.data;
      REPEAT UNTIL (cm.flag=CHR(0))
         CALL schedule(ABS(random) MOD 211); {wait a random time}
         COMCALL cm;
      END REPEAT;
      MOVE cm.data TO em.data;
      RETURN em;
   END REPEAT;

END PROCESS
```

D.6 Receiver

```
{process received messages- toss the bad ones, send broadcast to router,
 data to ARQ}
receiver: PROCESS

DECLARE
   cp: commess ACCEPTPORT;
   cm: commess;
   cim: cominitmess;
   em: ethermess;
   router: ethermess CALLPORT;
```

```
    arqp: ethermess CALLPORT;

 CREATE PHASE
    CONNECT cim.recp TO cp;
    cim.baud=baudrate(9600);
    cim.carriersense=20;
    COMCALL cim;
    CONNECT arqp TO "arqdistrib";
    CONNECT router TO "router";

 MAIN PHASE
    REPEAT
       WAIT cp;
       RECEIVE cm FROM cp;
       MOVE cm.data TO em.data;
       IF cm.flag=CHR(0) THEN (good reception)
          IF em.data[0]=CHR(0) THEN SEND em TO router; (broadcast)
          ELSE IF em.data[0]=nodeid THEN SEND em TO arqp; (for us)
          END IF; END IF;
       END IF;
    END REPEAT;

 END PROCESS
```

D.7 ARQ

```
(distribute packets to the proper ARQ process handling that link)
arqdistr: PROCESS

DECLARE
   ep: ethermess ACCEPTPORT;
   em: ethermess;
   arq: TABLESET (
          t: TABLE key(id) (
             id: CHAR;
             port: ethermess CALLPORT;
             cmp: COMPONENT);
          r: ROW IN t; );
   arqs: arqstart;
   c: CHAR;

CREATE PHASE
   PUBLISH ep AS "arqdistrib";

MAIN PHASE
   REPEAT
      WAIT ep;
      RECEIVE em FROM ep;
```

```
      c=em.data[1]; {the source}
      IF c=nodeid THEN c=em.data[0]; END IF; {a send message}
      DO
         FIND arq.r KEY(c);
      ON (not found) {start a new arq process for this link}
         ALLOCATE arq.r KEY(c);
         arqs.id=c;
         CREATE arq.r.cmp(arqs) MODULE("arq.cob");
         MOVE arqs.port TO arq.r.port;
      END DO;
      FORWARD em TO arq.r.port;
   END REPEAT;

END PROCESS


{does the arq for one link (i.e. one node pair) }
arq: PROCESS(arqs:arqstart)

DECLARE
   em: ethermess
   ep: ethermess ACCEPTPORT;
   router: ethermess CALLPORT;
   sender: ethermess CALLPORT;
   sendp: ethermess CALLPORT; {internal list of stuff to be sent}
   sendlist: ethermess ACCEPTPORT;
   current: ethermess; {message we are waiting for an ack on}
   ack: CHAR STRING;
   his,ours: BOOLEAN;
   timeout: long;

CREATE PHASE
   his=FALSE;
   ours=FALSE;
   ack=arqs.id!nodeid!(chr(0)!chr(0));
   CONNECT arqs.port TO ep;
   CONNECT sendp TO sendlist;
   CONNECT sender TO "sender";
   CONNECT router TO "router";

MAIN PHASE
   REPEAT
      IF length(ep)<>0 THEN
         RECEIVE em FROM ep;
         IF em.data[1]=nodeid THEN {message to be sent}
            SEND em TO sendp; {just queue it up}
         ELSE {received message}
            IF em.data[2]=CHR(ours) THEN {ack} DISCARD current; END IF;
            IF em.data[3]<>CHR(his) THEN {new reception}
               his=NOT his;
               SEND em TO router; {send to the higher level}
            END IF;
            IF empty(current) and length(sendlist)=0 THEN
               IF length(em.data)>4 THEN
```

```
                  {nothing else to send, so send an ack}
                  em.data=ack;
                  SEND em TO sendp;
               END IF;
            END IF;
         END IF;
      ELSE
         IF empty(current) THEN
            IF length(sendlist)=0 THEN SLEEP {nothing to do}
            ELSE {do sendlist}
               RECEIVE current FROM sendlist;
               ours= ours XOR length(current.data)>4;
               current.data[2]=CHR(his);
               current.data[3]=CHR(ours);
               CALL sender(current); {make sure we get the message back}
               timeout=time+3000+random&$0FFF; {approx .25 to .75 secs}
            END IF;
         ELSE
            IF time>=timeout THEN {have to try again}
               current.data[2]=CHR(his);
               current.data[3]=CHR(ours);
               CALL sender(current);
               timeout=time+5000+random&$1FFF; {approx .5 to 1.5 secs}
            ELSE SLEEP;
            END IF;
         END IF;
      END IF;
   END REPEAT;

END PROCESS
```

D.8 Router

```
{puts the immediate destination on messages, forwards them, broadcasts routes}
router: PROCESS

DECLARE
   em: ethermess;
   ep: ethermess ACCEPTPORT;
   arqp: ethermess CALLPORT;
   sender: ethermess CALLPORT;
   application: ethermess CALLPORT;
   hops,node: CHAR STRING;
   i,j: integer;
   timeout: long;
   c: CHAR;

CREATE PHASE
```

```
   PUBLISH ep AS "router";
   CONNECT arqp TO "arqdistrib";
   CONNECT sender TO "sender";
   CONNECT application TO "application";

MAIN PHASE
   zerostring(hops,6); {format is just like a broadcast message}
   zerostring(node,5);
   REPEAT FOR (i=2 TO 5) hops[i]=CHR(255); END REPEAT;
   hops[ORD(nodeid)+1]=0; {we are zero away from ourselves}
   node[ORD(nodeid)]=nodeid;
   timeout=time+10000;

   REPEAT REPEAT
     IF length(ep)<>0 THEN
        RECEIVE em FROM ep;
        IF em.data[1]=nodeid THEN {send this message from us}
           c=node[ORD(em.data[4])]; {destination node}
           IF c<>CHR(0) THEN
              em.data[0]=c;
              SEND em TO arqp; {send the message}
              LEAVE REPEAT;
           ELSE SEND em TO ep; {save it for later}
           END IF;
        ELSE {this message was received by us}
           c=em.data[1]; {immediate source}
           IF em.data[0]=CHR(0) THEN {a broadcast message}
              REPEAT FOR (i=2 TO 5)
                 j=ORD(em.data[i])+1; {number of hops to i}
                 IF j<ORD(hops[i]) THEN
                    hops[i]=CHR(j);
                    node[i-1]=c;
                    timeout=$80000000; {smallest number}
                 END IF;
              END REPEAT;
           ELSE {data message received}
              hops[ORD(c)+1]=1; {in case we don't know this guy yet}
              node[ORD(c)]=c;
              IF em.data[4]<>nodeid THEN {not for us}
                 em.data[1]=nodeid; {now from us}
                 SEND em TO ep; {forward it}
              ELSE {for us} SEND em TO application;
              END IF;
           END IF;
           LEAVE REPEAT;
        END IF;
     END IF;
     IF time>=timeout THEN {send broadcast again}
        em.data=hops;
        SEND em TO sender;
        timeout=time+20000;
     END IF;
     SLEEP; {give someone else a chance}
   END REPEAT; END REPEAT;
```

```
END PROCESS
```