499

The Crystal Nugget
Part I of the First Report on
The Crystal Project

Robert Cook

Raphael Finkel

David DeWitt

Lawrence Landweber

Thomas Virgilio

CONTENTS

1. Introduction to Crystal

The University of Wisconsin Crystal project was funded starting in 1981 by the National Science Foundation Experimental Computer Science Program to construct a multicomputer with a large number of substantial processing nodes. The original proposal called for the nodes to be interconnected using broadband, frequency-agile local network interfaces. Each node was to be a high performance 32 bit computer with a approximately 1 megabyte of memory and floating-point hardware. The total communications bandwidth was expected to be approximately 100 Mbits/second.

During the first year of the project, these specifications have been refined. We have decided to buy approximately 40 node machines, each a VAX-11/750. The interconnection hardware will be the Proteon ProNet. Currently, the ProNet is available in a 10 Mbits/second version. We have contracted with Proteon to increase the effective bandwidth to 80 MBits/second.

1.1. Software Overview

The purpose of this hardware is to promote research in distributed algorithms for a wide variety of applications. In order to provide different applications simultaneous access to the network hardware, we have designed a software package called the *nugget* that resides on each node. In brief, the nugget provides the following facilities:

1. The nugget enforces allocation of the network among different applications by virtualizing communications within partitions of the network. These partitions are established interactively through a host machine.
2. Backing store is shared among the nodes by nugget facilities to virtualize disks.
3. Interaction between the user and individual machines is provided by the nugget facility of virtual terminals.

4. Initial loading, control, and debugging of programs on node machines is controlled by nugget software.

The Charlotte operating system is designed to provide standard interactive operating system support within a Crystal partition. The Charlotte *kernel* provides

1. processes
2. multiprocessing
3. inter-process communication that hides node boundaries
4. mechanisms for scheduling, store allocation, and migration.

All policies in Charlotte are concentrated in *utility processes*. They are designed so that each such process controls a policy on its own set of machines. The set may range in size from one machine to the entire partition. The processes that control the same resource on different machine sets communicate with each other to achieve global policy decisions. The utilities that have been designed so far include a switchboard, a program starter, and the file server. In addition, there are non-policy utilities for command interpretation and program connection.

We expect that Crystal will be used for a wide range of applications. Currently research is underway in distributed operating systems, programming languages for distributed systems, tools for debugging distributed systems, multiprocessor database machines, parallel algorithms for math programming, numerical analysis and computer vision, and evaluating alternative protocols for high performance local network communications.

All Crystal software is being written in a local extension to Modula. Our compiler, which runs on a VAX running Berkeley 4.1 Unix, employs syntactic error correction through the FMQ algorithm and is quite fast. The code it generates compares well with that produced by the C compiler.

1.2. Phases of the project

The first phase of the project was dedicated to defining both the hardware and the software. This phase ended in December 1982. Decisions were reached concerning both the node machines and the interconnection devices. The node machine decision was difficult. We had to balance our concerns for reliability, availability, speed, and cost. The machine we chose, the VAX-11/750, although not as fast as others we investigated, had the advantage of being a known architecture for which our Modula compiler already generates code. The Proteon network is currently available. We have been using this network to interconnect our Unix VAX machines and have found it to be extremely reliable.

During the first phase, the nugget was specified and a prototype implementation was completed on a network of eight Digital Equipment PDP-11/23 computers connected by the Megalink CSMA broadband network manufactured by Computrol. Charlotte was also specified and the kernel debugged on this network.

The second phase of the project has just gotten underway. We are finalizing the nugget specifications, which changed in minor ways when we decided that the node machines would be VAXen. The nuggetmaster, which controls the partitions, has also been specified. Charlotte is undergoing debugging of the utility processes. During this phase, which lasts until July 1984, we will transfer the nugget and Charlotte to the node machines and modify them as necessary for the ProNet. Charlotte will be modified to fit with the nugget. (Until now, they have been developed independently.) The utility processes will be supplemented with login and authentication processes, and the file system will be converted to use Crystal disks instead of a file system on the host machine. We plan to have a production, stable operating system by the end of this phase.

The third phase of the project will see large-scale applications actively pursued. Some of this work will start during the second phase. We also expect to re-evaluate the hardware decisions at some point during this phase. There is some reason to expect that frequency-agile modems will be available that will make communication within each partition truly independent of communication within other partitions. Each partition will be able to use its own set of frequencies. Work with optical fiber technology for computer interconnection is also underway at various laboratories around the country. Within five years, impressive bandwidths should be available, reaching into the gigabit/second range. We will continue to monitor progress in this area.

1.3. This report

The purpose of this report is to describe the current state of the design and implementation of the Crystal project. It is intended for readers who have no familiarity with Crystal and wish to see the design decisions that have been made. It is also intended for implementers who need a coherent and reasonably complete specification in order to interface various parts of the project. This dual readership requires us to repeat ideas, first presenting them in an overview fashion, and then diving into tedious details. We urge the reader to skip over those parts of the document that are not at the right level of detail. This report is divided into several documents.

This document describes the nugget. After defining the technical terms that will be used throughout this report, we present the nugget specifications with regard to the interface between nugget and client, concentrating on communications, timing, and device services.

2. Definitions We will use the following terms throughout this document:

CLIENT: A program executing under the control of the nugget on a node. There is exactly one client per node.

DISKSERVER: A specialized client on a node not a member of any partition, used for actual disk access.

HOST: The computer connected to the user's terminal. There may be several hosts. They will typically be VAXen running Unix. Each host is connected to the Crystal communications medium.

NODE: A Crystal computer, connected to the Crystal communications medium. There will be many (on the order of 50) identical nodes. While this structure is being built, there will very likely be nodes of various versions (PDP-11/23's and VAX-11/750's).

NUGGET: A program that resides on each node. It virtualizes communications within each partition and with the nuggetmaster. It provides other services to the client.

NUGGETMASTER: A program invoked directly by a user. The nuggetmaster helps form partitions in the Crystal hardware and allows the user to control programs running within owned partitions. There may be several nuggetmasters running at any time on each host.

PARTITION: A subset of the nodes owned by a user.

USER: The human researcher, controlling work through a terminal on a host machine.

VIRTUAL DISK: A service provided by the nugget for the client. It emulates a disk by using part of a physical disk, possibly on another node machine.

VIRTUAL TERMINAL: A service provided by the nugget and nuggetmaster that allows both input and output to be directed from the client to the user's physical terminal on the host. The user's physical terminal may be multiplexed among many virtual terminals, and buffering is provided.

3. Client-Nugget Interaction

The n nodes in a partition are numbered logically 1 to n, in an order determined by the nuggetmaster command the user employed to build the partition. In addition, each partition has a logical node 0, which is mapped to a program running on the host.

The nuggetmaster binds the client and nugget into a single load module for each node. The logical-to-physical node address table used by the nugget is also linked into the nugget as part of that load module. Tables needed for virtual disks and virtual terminals (discussed below) are also linked in at this time.

Definition files that document common structures used by the client and the nugget are stored in the directories "/usr/crystal/nugget/*_include" on each host. There is a directory for each language supported. This list includes Modula, and will include C and StarMod. We will present parts of the Modula version of the definition files throughout this document. A complete listing of the Modula version can be found in the last chapter of this document. The definition file lists variables that are imported and exported by the nugget:

```
-------------------------------------------------------------------
module nugget;

define (* interface routines *)
        NuggetSend, NuggetReceive, (* communication routines *)
        DiskOp, Get_Disk_Info,          (* virtual disk routines *)
        EnableInput, OutputReady,  (* virtual terminal routines *)
        Pause, Save_Nugget_State;  (* debug routines *)

use (* parameters *)
        CommDevice, CommIntVectors, (* communication *)
        TimeDevice, TimeIntVector,  (* virtual clock *)
        _client,           (* client entry point after nugget initialization *)
        MaxLogicalAddr, SelfLogicalAddr, (* partition data *)
        No_of_Disks, DiskDevices, DiskIntVectors, (* virtual disk *)
        No_of_Terms, TermDevices, TermIntVectors; (* virtual terminal *)
-------------------------------------------------------------------
```

The nugget requires memory mapping to be enabled and runs in kernel mode. It initializes the kernel mapping table and enables memory mapping. It is allowed to use any pages it wishes for its own purposes, so long as this use is invisible to the client. The client's stack is kept inviolate; hence the nugget uses its own stack. However, it switches to the client's stack whenever the client is active.

The nugget provides many services to the client and the partition. These services are machine initialization, communications between node machines, a virtual clock, access to disks shared with other partitions (virtual disks), a simple debugger, and multiplexing of all partition terminals to the user's terminal on the host (virtual terminals). Most services appear to the client as devices. Other physical devices connected to the node may be treated by the client in whatever fashion the user wishes without danger of interference by the nugget.

Each service is introduced in this chapter and then detailed more fully in its own chapter. That part of the definition file that pertains to each service is introduced in each chapter.

The client invokes most services by subroutine call, implicitly passing the appropriate device registers. These subroutine calls effect a "start I/O" operation on the device. On the VAX node machines these service calls must be use the CALLS instruction. These calls return with a result in register 0. By convention, a result of 0 usually means the request was legal and -1 means it was illegal. The client may immediately modify device registers after the call returns, because the nugget copies all relevant information before returning.

At the completion of a legal operation, the client will be notified by an interrupt call through the appropriate interrupt vector. Interrupt vectors should not be modified by the client while an operation on the associated device is in progress. The client can learn the details of the operation by inspecting the device registers or parameters pushed on the stack. The client interrupt handler should return via the REI instruction (on VAX nodes) after popping any parameters from the stack. The client's interrupt stack is the kernel stack and CPU priority during interrupts is DEVICEPRIORITY (except for the clock handler, which runs at CLOCKPRIORITY). It is up to the client to save the state of the machine upon interrupt.

3.1. Initialization

The first service that the nugget performs is to initialize the machine to a known state. Initialization happens before the client code is entered and is not started by subroutine call. During initialization the nugget resets all I/O devices, initializes the System Control Block, sets aside space for the nugget's stack, zeros out the nugget and client bss (uninitialized data) segment, initializes the system page table (identifying virtual and physical addresses), enables memory management, initializes its own internal data, disables the client's virtual clock (described below), defines a process context at the top of the nugget's stack space setting the kernel stack to the address supplied by the client, and lastly transfers control to the client at label "_client" (in Modula) with high priority.

The client should not return from _client. The client should initialize all device registers and interrupt vectors before lowering priority or calling nugget service routines. A complete description of the initialization phase can be found in /usr/crystal/nugget/VAX_startup/README.

3.2. Communication

The nugget provides communication with other nodes of the partition. Communication is by messages. The nugget controls all communications between nodes of the partition and the host machines. A client is allowed to communicate with members of the client's partition only. Each partition is protected from stray messages from other partitions. The nugget provides two modes of service: datagram and reliable in-order delivery.

To send a message, the client places a message in a buffer that it supplies, describes the buffer and the node address of the destination in the send registers, and calls the procedure NuggetSend. A completing send passes back the success of the operation and the destination address. Many sends may be in progress at one time, but only one per destination node. In addition, the client may broadcast one message to all nodes in its partition.

To receive a message, the client describes in the receive register the buffer it supplies and calls NuggetReceive. A completing receive request passes back the status of message, the source's node address, size of the message, and mode of communication. Only one receive may be active at one time.

For both send and receive, the client refers to other nodes only through logical node addresses. These must be in the interval NUGGETMASTER to MaxLogicalAddr: NUGGETMASTER (constant 0) is the logical address of the nuggetmaster; MaxLogicalAddr is the number of the nodes in the partition. In addition the variable SelfLogicalAddr tells the client the logical node address of its machine. The nugget initializes

MaxLogicalAddr and SelfLogicalAddr during the initialization phase.

The client and nugget exchange information about messages in two device registers, SendRegister and ReceiveRegister, which are structured data areas imported by the nugget. A message consists of a header composed by the nugget and a body composed by the client. The body may be divided into several (up to MAXMESSPARTS) disjoint regions, each contiguous in kernel virtual space. A message body is described by the BufferDescrip field of the SendRegister or ReceiveRegister. BufferDescrip is an array describing all the parts of a client's message's body. Each array entry describes one region by a start address and length. A length of 0 indicates a missing region. The total length of all regions must lie between MINMESSLENGTH and MAXMESSLENGTH.

3.3. Clock

The nugget provides a timing service called the *virtual clock*. This service is continuous and not invoked by a procedure call. The virtual clock consists of a data area supplied by the client. It contains a field called Count, which the client may set and inspect at will. Every tick of the virtual clock (currently, 60th of a second) causes Count to be decremented. Count may become negative. If the count becomes 0, then the client is interrupted at CPU priority CLOCKPRIORITY.

3.4. Disk

Virtual disks are created and destroyed by users interacting with the nuggetmaster. A user who builds a partition may specify that certain virtual disks be made part of that partition. Each virtual disk is emulated by some physical disk, attached to some physical node. One physical disk may emulate many virtual disks. If the physical node associated with a virtual disk is part of a user's partition, then no other users have access to virtual disks on the same physical disk. This association allows the client to make reliable timing measurements of disk operations. Alternatively, the virtual disk's node may be omitted from the partition. In this case, the clients in the partition still have access to the virtual disk, but other partitions may use the same physical disk at the same time. We call the node connected to a shared physical disk outside of any partition a *diskserver*.

The client can only access a virtual disk through the nugget, so virtual disks that are not in the partition are protected from tampering. The nugget on each node in a partition has access to all virtual disks in the partition. The user may of course decide to submit virtual disk access requests from only one client.

Each virtual disk is assigned a number in the range 0 to No_of_Disks, a variable imported and initialized by the nugget. Cylinders on the disk are referred to by logical cylinder numbers. Each virtual disk has a set of registers declared in the definition file and imported by the nugget. These registers provide communication between the client and the nugget.

The nugget uses the registers to describe the parameters of the disk, including information on number of cylinders, surfaces per cylinder, and sectors per surface. If the physical disk is connected to this node, then the nugget also provides information on the current position of the disk arm.

The client uses the registers to specify the type of operation ("read" or "write"), the logical cylinder, the surface, the sector, the number of sectors, and the kernel virtual address of the buffer to be used in the operation. Operations are started by calling the procedure DiskOp, passing as a parameter the name of the virtual disk involved in the operation. The client is informed by an interrupt when the operation has completed. In order to complete the operation, the nugget may have to exchange messages with the nugget on the node that is connected to the associated physical disk.

3.5. Terminals

The user may request that each machine be provided with a number of *virtual terminals* to be multiplexed to the user's terminal on the host. The user can then interact with one physical terminal and view or control the progress of the entire partition. Control of this "multi-terminal" is provided by commands to the nuggetmaster, which are detailed in the nuggetmaster chapter.

The number of virtual terminals on each node is known to the client in the variable No_of_Terminals, imported and initialized by the nugget. Each terminal is referred to by a number in the range 0 to No_of_Terminals-1. Each node has at least one virtual terminal, the *console*, numbered 0. Each terminal has its own set of registers. All registers are exported by the client in the array TermDevices.

To write to a terminal, the client places the character to be written in Output_Char of the terminal's register and calls OutputReady, passing the name of the virtual terminal. When the terminal can accept new output (the last character has been sent to the nuggetmaster) the client is notified by interrupt. The nuggetmaster may buffer the character until it can be displayed. To read from the terminal, the client calls EnableInput, passing the name of the virtual terminal. The client is notified by interrupt when a character arrives from the nuggetmaster.

3.6. Debugger

Lastly, the nugget provides a simple synchronization and debugging service to the client:

```
procedure Pause; external;
```

A client that invokes Pause does not continue until the pause is released by the nuggetmaster. No interrupts of any kind are seen by the client during pause. The nugget remains active during the pause state and continues to send and receive messages. Notification of successful operations is given after the pause state ends. Of course, the client is unable to submit new requests while pausing. The client's virtual clock is not decremented in pause state. During pause, many debugging and partition-control commands can be executed through the nuggetmaster and are outlined in a later chapter.

The nugget also provides a simple means for debugging itself:

```
const
            NUGSTATESIZE = 500;
type
            NuggetState = array 0:NUGSTATESIZE-1 of shortint;
procedure Save_Nugget_State(var SavedState : NuggetState);
            external;
```

The procedure Save_Nugget_State raises priority to that of the clock and copies all data used by the nugget to SavedState. These data include the state of the communication device. The saved state may be inspected to determine unexplained actions.

4. Communication Interface

The nugget controls all communications among clients residing in the nodes of a partition. Communication is by messages. The communication service appears to the client as a device that sends and receives messages. Relevant declarations can be found in this chapter and in the chapter detailing the Modula definition file at the end of this document.

The client can send and receive messages by calling NuggetSend and NuggetReceive. The client implicitly passes parameters describing the operation in the device registers SendDevice and ReceiveDevice. These devices are part of CommDevice, a data area exported by the client and imported by the nugget. When a communication operation completes, the client is notified by interrupt at CPU priority level DEVICEPRIORITY through SendIntVect and ReceiveIntVect. These two vectors should be initialized by the client before lowering CPU priority in the client's initialization process.

The client can choose between two modes for message delivery. "Datagram" mode writes the message out on the line. The nugget does not guarantee its delivery but will make a good faith effort to get it to the nugget of the other machine. "Ack-receipt" mode is a reliable delivery service. The nugget will not only deliver the message to the other nugget, but will wait for notification by the other nugget that its client accepted the message. The client is informed if delivery to the other client is not possible.

The client refers to other nodes through logical node addresses. These must be in the interval NUGGETMASTER to MaxLogicalAddr: NUGGETMASTER (constant 0) is the logical address of the nuggetmaster, and MaxLogicalAddr is the number of the nodes in the partition. The client's own machine's logical name can be found in SelfLogicalAddr. These two variables are supplied by the client and initialized by the nugget.

A message consists of a header composed by the nugget and a body composed by the client. The body may be divided into several (up to MAXNOMESSPARTS) disjoint regions, each contiguous in kernel virtual space. A message body is described by the BufferDescrip field of the SendRegister or ReceiveRegister. BufferDescrip is of type MessBodyDescrip, which is an array describing all the parts of a client's message's body. Each array entry describes one region by the address of the start of the region (BodyAddr) and the length in bytes of the region (Length). A length of 0 means that the BodyAddr field is not valid, and that this region is missing. Length must be an even number of bytes. The sum of all lengths must lie between MINMESSLENGTH and MAXMESSLENGTH.

The SendRegister fields are interpreted as follows:

```
type
    InterruptVector =
        record
            InterruptHandler : KernelAddr;
            NewPSW : midint;
        end; (* InterruptVector *)
    PartDescrip =
        record (* describes one part of a message body *)
            BodyAddr : KernelAddr;
            Length   : longint;
        end; (* PartDescrip *)
    MessBodyDescrip = array 0:MAXMESSPARTS-1 of PartDescrip;

var
    CommIntVectors :
        record  (* the communication interrupt vectors *)
            SendIntVect : InterruptVector;
                (* client initializes to interrupt handler's
                address and desired lower half of PSW upon
                interrupt *)
            ReceiveIntVect : InterruptVector;
                (* not used in sending *)
        end; (* CommIntVectors *)
    SendRegister :
        record (* client sets all fields *)
            BufferDescrip : BodyDescrip;
                (* points to each part of message *)
            Mode : (DataGram,AckReceipt);
                (* desired communication circuit *)
            DestAddr : LogicalNodeAddr;
                (* destination's logical node address *)
        end; (* SendRegister *)

procedure NuggetSend : StandardInt; (* sends SendRegister's message *)
```

To send a message, the client first sets the fields of SendRegister. BufferDescrip should point to the message buffer, with each array entry's kernel virtual address (BodyAddr) and length in bytes (Length) corresponding to the parts of the message buffer. Mode and destination address (DestAddr) must also be set. The client then calls NuggetSend.

NuggetSend returns the standard legal/illegal flag. A call to NuggetSend is illegal if a pending send to the same destination node has not yet been dispatched and acknowledged. The client may have MaxLogicalAddr sends pending. The client may send to the nuggetmaster, logical machine 0, but not to itself.

When the send completes the client will be interrupted at CommIntVectors.SendIntVect.InterruptHandler with the lower 16 bits of the PSW as set in CommIntVectors.SendIntVect.NewPSW. These interrupt fields are typically set by the client at initialization time. Changes to these fields during a send operation will produce unpredictable results.

If client the Mode is AckReceipt, the send cannot complete successfully unless the message is acknowledged. Completion may mean that the nugget timed out trying to send and that the message was not received. At completion of the send the nugget pushes onto the stack either 0 or 1 to indicate that the send operation succeeded (0) or timed out (1) and then pushes the destination's logical node address. These two parameters should be popped from the stack before the client returns from the interrupt. There is no ready bit or interrupt-enable bit in SendRegister; client-notification interrupts are always enabled.

In addition to sending a message to a single remote machine, the client may send a single broadcast message to all nodes in the partition. The nugget does not deliver broadcast messages to the sending client. A broadcast message must be a datagram. In addition to one send directed to every other node in the partition, the client may have one broadcast request active. To send a broadcast message the client specifies BROADCAST (value -1) as the node address when requesting a send. Broadcast messages are received as if they were a node-to-node datagram.

The fields in ReceiveRegister are interpreted as follows:

```
------------------------------------------------------------------
var
        CommIntVectors :
                record  (* communication interrupt vectors *)
                        SendIntVect : InterruptVector; (* not used in receive *)
                        ReceiveIntVect : InterruptVector;
                                (* client sets to virtual address of interrupt
                                handler, and desired lower 16 bits of PSW upon
                                interrupt *)
                end; (* CommIntVectors *)
        ReceiveRegister :
                record
                        (* the following are set by the client *)
                        BufferDescrip : BodyDescrip;
                                (* client sets to point to all parts of buffer *)
                        (* the following are set by the nugget *)
                        Status : (MessArrived, NoMessTimeOut);
                                (* type of interrupt *)
                        SourceAddr : LogicalNodeAddr;
                                (* source's logical node address *)
                        Mode : (DataGram,AckReceipt);
                                (* communication circuit of message *)
                        Length : longint;
                                (* total length of arrived message *)
                end; (* ReceiveRegister *)

procedure NuggetReceive : StandardInt;
        (* enables receipt of message *)
------------------------------------------------------------------
```

To receive a message, the client sets the BufferDescrip field of ReceiveRegister to point to the message buffer by setting each array entry's virtual address (BodyAddr) and length in bytes (Length) to those values corresponding to the parts of the message buffer. The buffer should be large enough to hold any expected message. Each part will be filled to capacity by the incoming message before the next part is used. Messages that do not fit into the buffer will not be accepted or acknowledged. The sum of all Lengths must be at least MINMESSLENGTH. The receiving client does not specify which client it wishes to receive from.

NuggetReceive returns the standard legal/illegal flag. A call to NuggetReceive is illegal if a pending receive has not yet been fulfilled. Several sends and one receive may be outstanding at the same time.

When a message is accepted by the nugget from another client, the client will be interrupted at CommDevice.ReceiveIntVect.InterruptHandler with the lower 16 bits of PSW as set in CommDevice.ReceiveIntVect.NewPSW. These interrupt fields are typically set by the client at initialization time. Changes to these fields during a receive operation will produce unpredictable results.

The client's receive request may be interrupted by a timeout that indicates that no messages from any other node has arrived. The nugget sets ReceiveRegister.Status to distinguish between the receipt of a message (MessArrived) and the timeout (NoMessTimeOut). If Status is MessArrived, SourceAddr will be set to the source's logical node address, Mode will be set to the mode of the newly arrived message, and Length will be set to the total length of the message. The priority level of the processor upon interrupting the client will be the priority of the device (DEVICEPRIORITY). There is no ready bit or interrupt-enable bit in ReceiveRegister. Client notification interrupts are always enabled.

The entire communications definition file is reproduced at the end of this document.

5. Virtual Clock Interface

The declarations for the virtual clock are as follows:

```
--------------------------------------------------------------------
const
        MIN_COUNT = -32767; (* Count will not fall below this value *)
        CLOCKPRIORITY = 24; (* clock priority level *)

var
        TimeIntVector : InterruptVector;
        TimeDevice =
                record
                        Count : longint;
                end; (* TimeDevice *)
--------------------------------------------------------------------
```

The client may set or inspect the Count field at will. Every tick (currently, 60th of a second), the nugget decrements Count. The count may become negative. The nugget will not decrement Count if so doing would change it from negative to positive. Count will therefore not fall below MIN_COUNT. If the count becomes 0, then the client is interrupted at location TimeIntVector.InterruptHandler with the lower 16 bits of the PSW set to TimeIntVector.NewPSW. These fields should be set by the client at initialization time. Changes to these fields when the client clock is positive will produce unpredictable results. The priority level will be that of the clock (CLOCKPRIORITY).

The client may change Count at any time. To change the interrupt vector, the client should first set Count to any negative integer, disabling the clock interrupt. Count is initialized by the nugget to MIN_COUNT.

6. Virtual Disk Interface

The virtual disk program presents a disk device to the client. The program controls access to the disk, thereby protecting the client's data from access by other partitions over time and preventing the client from accessing disk data belonging to other partitions. Virtual disks are created by users interacting with the nuggetmaster. A user may include access to one or more virtual disks in a partition. Each node in the partition may access those virtual disks. Virtual disk names are uniform across the partition. The definition file contains a section defining the virtual disk interface. This part of the include file can be found at the end of this chapter.

The client refers to each virtual disk by its logical disk number. The maximum such reference is in No_of_Disks, which is imported and initialized by the nugget. All cylinder references are logical numbers as well. Surface and sector numbers are physical.

Each virtual disk has its own set of device registers. These are accessed through the array DiskDevice. Each register has three sets of fields. First, the static fields, initialized by the nugget, describe the device (Max_Logical_Cylinder, Num_Surfaces, Num_Sectors_Per_Track, and SectorLength) and should not be written by the client.

Second, the dynamic fields describe the position of the disk arm (Current_Cylinder) and the status of the I/O operation (Status_of_I_O). These fields are updated by the nugget during I/O operations and should not be written by the client. If the associated physical disk is not attached to this node, Current_Cylinder contains the constant NOTDEFINED. Status_of_I_O contains the status of the last completed I/O operation. It is reset with each call to the procedure DiskOp. The status can indicate two failure modes: An actual disk operation failed (DiskFailure), and there was a possible failure of the I/O operation (MessFailure). In the second case, communication with a remote disk failed; the I/O operation may have succeeded.

Finally, command fields describe the I/O operations. This information is written by the client before calling DiskOp. "Operation" indicates which operation is requested. "Cylinder", "Surface" and "Sector" specify the starting position of the I/O. A seek operation to that track is implied. "Sector_Count" specifies the number of contiguous sectors to be transferred. Sector_Count must not exceed Num_Sectors_Per_Track. "BufferAddr" holds the starting address of the buffer used for the transfer. The buffer must be virtually contiguous in kernel space.

To schedule a disk operation, the client sets all the fields of RequestInfo and then calls DiskOp, passing the logical disk number of the virtual disk. DiskOp returns the usual legality indication. DiskOp can fail if the disk name is out of range, Request_Info is illegal, or the previous disk operation has not yet completed. A legal call will schedule the disk operation.

Upon completion of the disk operation, the client is interrupted at the address stored in DiskIntVector. The virtual disk number is pushed onto the stack as a longint prior to the interrupt. This parameter should be removed before the return from interrupt. The interrupt routine will run at the priority level of the virtual disk (DEVICEPRIORITY). This priority level is the same as the communication device. The lower 16 bits of the processor status word is set to NewPSW of the interrupt vector prior to the interrupt. It is the client's responsibility to save the machine state upon interrupt.

The disk interface section of the definition file follows:

```
---------------------------------------------------------------------
const
        MAXDISKS = ? (* max number of virtual disks per machine *)
        NOTDEFINED = -1; (* position information not available *)

type
        OpInfo =  (* command fields of a device register *)
                record
                        (* these fields are client read/write *)
                        Operation : (Read_Disk, Write_Disk);
                        Cylinder : longint; (* logical cylinder (track) *)
                        Surface  : longint; (* physical surface *)
                        Sector   : longint; (* physical sector *)
                        Sector_Count : longint; (* number of sectors to
                                transfer *)
                        BufferAddr : KernelAddr; (* kernel virtual address
                                of start of data *)
                end; (* OpInfo *)
        DiskRegisters =
                record
                (* static fields, initialized by nugget, read by client *)
                        Max_Logical_Cylinder : longint;
                        Num_Surfaces : longint;
                        Num_Sectors_Per_Track : longint;
                        SectorLength : longint; (* number of bytes per disk
                                sector *)
                (* dynamic fields, set by nugget, read by client *)
                        Current_Cylinder : longint; (* current position of
                                disk arm or NOTDEFINED *)
                        Status_of_I_O : (Success, DiskFailure, MessFailure);
                (* command fields, set by client, read by nugget *)
                        RequestInfo : OpInfo;
                end; (*DiskRegisters*)

var
        No_of_Disks : longint; (* actual number of virtual disks,
                bounded by MAXDISKS, initialized by nugget *)
        DiskIntVectors : array 0:MAXDISKS-1 of InterruptVector;
        DiskDevice : array 0:MAXDISKS-1 of DiskRegisters;

procedure DiskOp(DiskNo : longint) : StandardInt; external;
        (* schedules a virtual disk operation *)
```

7. Virtual Terminal Interface

Virtual terminal service allows each client in a partition to access a terminal (for both input and output) that is multiplexed to the user's terminal on the host. The definition file contains a description of the virtual terminal interface. This interface can be found at the end of this chapter.

The virtual terminal "device" is controlled by nugget routines. Many of these routines communicate with the nuggetmaster's virtual terminal handler. The nuggetmaster specification details the commands that the user may employ to control the multiplexed physical terminal.

When a partition is specified by the user, each node is given one or more virtual terminals logically numbered 0 to No_of_Terminals-1. The variable No_of_Terminals is imported and initialized by the nugget. Terminal 0 is always present and acts as the node's console.

The virtual terminal appears to the client as a device. The client must initialize the device by setting the interrupt vectors. When a character has appeared on the input device, the client is notified by interrupt at InputIntVect.InterruptHandler at the priority of the terminal (DEVICEPRIORITY) providing that the client has called EnableInput on this device. The terminal priority is the same as that of the communication virtual device. Before calling the client's interrupt handler, the nugget pushes the virtual terminal number on the stack as a longint. This parameter should be removed by the client before returning from the interrupt.

When a character has been written by the output device, the client is notified at OutputIntVect.InterruptHandler. Again, the virtual terminal number is pushed on the stack prior to the call. For both interrupts, the lower 16 bits of the processor status word are set to NewPSW of the interrupt vector.

To read a character, the client calls EnableInput, passing the terminal number as a longint parameter. EnableInput returns a legality indicator. The Input_Status of the terminal is Device_Busy until the character arrives. After the interrupt notification, the client can read Input_Char. The client can examine Input_Status to see if the input character has been lost due to device errors.

To write a character, the client should set write the character to Output_Char of the appropriate entry in TermRegs, and call OutputReady, passing it, as a longint parameter, the virtual terminal number. OutputReady returns the usual legality indicator. After the character is written to the device (that is, sent to the nuggetmaster), the Output_Status of this device is set, the virtual terminal number is pushed on the stack, and the client is notified through its interrupt handler. This parameter should be removed before returning from interrupt. If the line to the nuggetmaster goes down, a value of DeviceError appears in the status field.

Flow control between the nugget and the physical terminal on the host is managed by a permission scheme whereby each end sends permissions to the other for characters. If the client tries to write to the virtual terminal, but the nugget does not have permission to send the character to the nuggetmaster, the output-completion interrupt is delayed until the character can be sent. If the user enters characters on the physical terminal that the nuggetmaster does not have permission to send, some characters are buffered. If the buffer is exceeded, no more characters are accepted. It is useful for the client to echo input so the user can tell how much has been accepted. See the nuggetmaster specifications for details.

The client/virtual terminal interface follows:

```
-------------------------------------------------------------------------
const
        MAXTERMINALS = ? (* maximum number of terminals per machine *)

type
        Status_Values = (DeviceBusy, CharOkay, DeviceError)
        TerminalRegisters =
                record
                        (* the following are client read only *)
                        Input_Char : char;  (* terminal input character *)
                        Input_Status : Status_Values; (* stat input stream *)
                        Output_Status : Status_Values; (* stat output stream *)
                        (* the following are client write only *)
                        Output_Char : char;  (* terminal output character *)
                end; (* TerminalRegisters *)

var
        TermIntVectors :
                record
                        InputIntVect  : InterruptVector;
                        OutputIntVect : InterruptVector;
                end; (* TermIntVectors *)
        No_of_Terminals : longint; (* actual number of terminals on node *)
        TermDevices : array 0:MAXTERMINALS of TerminalRegisters;

procedure EnableInput(TerminalNumber : longint) : StandardInt;
        (* client call to enable next input from TerminalNumber *)
        external;

procedure OutputReady(TerminalNumber : longint) : StandardInt;
        (* client call to write Output_Char of
                TermRegs[TerminalNumber] *)
        external;
-------------------------------------------------------------------------
```

8. Node Operation and Control

The node may be in one of several states. The nuggetmaster can cause the node to change states by sending commands to the nugget. The client can also cause the node to change states. The nugget sends a status message to the nuggetmaster upon state change. The set of states is as follows:

1. BOOT: A bootstrap program (very likely implemented on the boot tape) waits for a load image from any nuggetmaster. That image contains a nugget bound to a client. The nugget's tables already indicate the nature of

the partition. The bootstrap program transfers control to a fixed location in the nugget. The nugget, after initializing itself, transfers control to the client (at a fixed symbolic location: _client) and enters RUNNING state.

2. HALT: An actual 'halt' instruction has been executed. The node must be rebooted to do anything. In this state, interrupts are ignored. The purpose of this state is for failure insertion (and completeness, since the client may execute a halt instruction).

3. PAUSE: The user or client has requested a pause. The nugget masks all interrupts to the client and busy-waits on the communication device for a CONTINUE order from the nuggetmaster. The client's clock is frozen (that is, not decremented). The nugget remains active and continues to process outstanding client send and receive requests as well as outstanding virtual disk requests, but delays notifying the client about any completing operation until the CONTINUE order arrives. The nugget does not accept input characters from virtual terminals during the PAUSE state. When the CONTINUE order arrives, the nugget and client resume RUNNING state.

4. RUNNING: After a successful boot, the nugget and client are running normally.

The nuggetmaster provides the user several commands that modify the node's state as well as commands for simple debugging. The user's commands are implemented through messages from the nuggetmaster to the nugget. The commands that the nuggetmaster may give the nugget follow:

1. REBOOT: Returns node to the boot state. This order is legal in any state.

2. RESTART: Returns the client to the running state at symbolic location _client. This order is legal in RUNNING or PAUSE states. The order is obeyed following completion of all outstanding client requests. During this delay, the node runs in the PAUSE state. Users who intend to restart their

clients should take pains to make them re-entrant. In particular, initialized data must be re-initialized on every entry to __client.

3. HALT: Puts the node in the halt state. This order is legal in RUNNING or PAUSE states. Outstanding client requests are not honored by the nugget.

4. HELLO: This order is ignored by the nugget, but since every message from the host is acknowledged, it can be used to check if the nugget is still alive. This order is legal in RUNNING or PAUSE states.

5. PEEK: The order includes a 32-bit physical address. The nugget returns the 16 bits contained in the word at that address to the nuggetmaster. This order is legal in RUNNING or PAUSE states.

6. POKE: The order includes a 32-bit physical address and a 16-bit value to store. The nugget stores that value in the word at that location. This order is legal in RUNNING or PAUSE states.

7. CONTINUE: Moves the nugget from the pause state to the running state. This order is legal in the PAUSE state only. It is ignored in the RUNNING state. The client is notified of any requests that completed in the PAUSE state.

8. PAUSE: Puts the nugget and client in the pause state. This order is legal in the RUNNING state only. This command has the same effect as the client calling the procedure Pause.

9. Definitions We will use the following terms throughout this document:

CLIENT: A program executing under the control of the nugget on a node. There is exactly one client per node.

DISKSERVER: A specialized client on a node not a member of any partition, used for actual disk access.

HOST: The computer connected to the user's terminal. There may be several hosts. They will typically be VAXen running Unix. Each host is connected to the Crystal communications medium.

NODE: A Crystal computer, connected to the Crystal communications medium. There will be many (on the order of 50) identical nodes. While this structure is being built, there will very likely be nodes of various versions (PDP-11/23's and VAX-11/750's).

NUGGET: A program that resides on each node. It virtualizes communications within each partition and with the nuggetmaster. It provides other services to the client.

NUGGETMASTER: A program invoked directly by a user. The nuggetmaster helps form partitions in the Crystal hardware and allows the user to control programs running within owned partitions. There may be several nuggetmasters running at any time on each host.

PARTITION: A subset of the nodes owned by a user.

USER: The human researcher, controlling work through a terminal on a host machine.

VIRTUAL DISK: A service provided by the nugget for the client. It emulates a disk by using part of a physical disk, possibly on another node machine.

VIRTUAL TERMINAL: A service provided by the nugget and nuggetmaster that allows both input and output to be directed from the client to the user's physical terminal on the host. The user's physical terminal may be multiplexed among many virtual terminals, and buffering is provided.

10. Client-Nugget Interaction

The n nodes in a partition are numbered logically 1 to n, in an order determined by the nuggetmaster command the user employed to build the partition. In addition, each partition has a logical node 0, which is mapped to a program running on the host.

The nuggetmaster binds the client and nugget into a single load module for each node. The logical-to-physical node address table used by the nugget is also linked into the nugget as part of that load module. Tables needed for virtual disks and virtual terminals (discussed below) are also linked in at this time.

Definition files that document common structures used by the client and the nugget are stored in the directories "/usr/crystal/nugget/*_include" on each host. There is a directory for each language supported. This list includes Modula, and will include C and StarMod. We will present parts of the Modula version of the definition files throughout this document. A complete listing of the Modula version can be found in the last chapter of this document. The definition file lists variables that are imported and exported by the nugget:

```
-----------------------------------------------------------------
module nugget;

define (* interface routines *)
        NuggetSend, NuggetReceive, (* communication routines *)
        DiskOp, Get_Disk_Info,      (* virtual disk routines *)
        EnableInput, OutputReady,  (* virtual terminal routines *)
        Pause, Save_Nugget_State;  (* debug routines *)

use (* parameters *)
        CommDevice, CommIntVectors, (* communication *)
        TimeDevice, TimeIntVector,  (* virtual clock *)
        _client,        (* client entry point after nugget initialization *)
        MaxLogicalAddr, SelfLogicalAddr, (* partition data *)
        No_of_Disks, DiskDevices, DiskIntVectors, (* virtual disk *)
        No_of_Terms, TermDevices, TermIntVectors; (* virtual terminal *)
-----------------------------------------------------------------
```

The nugget requires memory mapping to be enabled and runs in kernel mode. It initializes the kernel mapping table and enables memory mapping. It is allowed to use any pages it wishes for its own purposes, so long as this use is invisible to the client. The client's stack is kept inviolate; hence the nugget uses its own stack. However, it switches to the client's stack whenever the client is active.

The nugget provides many services to the client and the partition. These services are machine initialization, communications between node machines, a virtual clock, access to disks shared with other partitions (virtual disks), a simple debugger, and multiplexing of all partition terminals to the user's terminal on the host (virtual terminals). Most services appear to the client as devices. Other physical devices connected to the node may be treated by the client in whatever fashion the user wishes without danger of interference by the nugget.

Each service is introduced in this chapter and then detailed more fully in its own chapter. That part of the definition file that pertains to each service is introduced in each chapter.

The client invokes most services by subroutine call, implicitly passing the appropriate device registers. These subroutine calls effect a "start I/O" operation on the device. On the VAX node machines these service calls must be use the CALLS instruction. These calls return with a result in register 0. By convention, a result of 0 usually means the request was legal and -1 means it was illegal. The client may immediately modify device registers after the call returns, because the nugget copies all relevant information before returning.

At the completion of a legal operation, the client will be notified by an interrupt call through the appropriate interrupt vector. Interrupt vectors should not be modified by the client while an operation on the associated device is in progress. The client can learn the details of the operation by inspecting the device registers or parameters pushed on the stack. The client interrupt handler should return via the REI instruction (on VAX nodes) after popping any parameters from the stack. The client's interrupt stack is the kernel stack and CPU priority during interrupts is DEVICEPRIORITY (except for the clock handler, which runs at CLOCKPRIORITY). It is up to the client to save the state of the machine upon interrupt.

10.1. Initialization

The first service that the nugget performs is to initialize the machine to a known state. Initialization happens before the client code is entered and is not started by subroutine call. During initialization the nugget resets all I/O devices, initializes the System Control Block, sets aside space for the nugget's stack, zeros out the nugget and client bss (uninitialized data) segment, initializes the system page table (identifying virtual and physical addresses), enables memory management, initializes its own internal data, disables the client's virtual clock (described below), defines a process context at the top of the nugget's stack space setting the kernel stack to the address supplied by the client, and lastly transfers control to the client at label "_client" (in Modula) with high priority.

The client should not return from _client. The client should initialize all device registers and interrupt vectors before lowering priority or calling nugget service routines. A complete description of the initialization phase can be found in /usr/crystal/nugget/VAX_startup/README.

10.2. Communication

The nugget provides communication with other nodes of the partition. Communication is by messages. The nugget controls all communications between nodes of the partition and the host machines. A client is allowed to communicate with members of the client's partition only. Each partition is protected from stray messages from other partitions. The nugget provides two modes of service: datagram and reliable in-order delivery.

To send a message, the client places a message in a buffer that it supplies, describes the buffer and the node address of the destination in the send registers, and calls the procedure NuggetSend. A completing send passes back the success of the operation and the destination address. Many sends may be in progress at one time, but only one per destination node. In addition, the client may broadcast one message to all nodes in its partition.

To receive a message, the client describes in the receive register the buffer it supplies and calls NuggetReceive. A completing receive request passes back the status of message, the source's node address, size of the message, and mode of communication. Only one receive may be active at one time.

For both send and receive, the client refers to other nodes only through logical node addresses. These must be in the interval NUGGETMASTER to MaxLogicalAddr: NUGGETMASTER (constant 0) is the logical address of the nuggetmaster; MaxLogicalAddr is the number of the nodes in the partition. In addition the variable SelfLogicalAddr tells the client the logical node address of its machine. The nugget initializes MaxLogicalAddr and SelfLogicalAddr during the initialization phase.

The client and nugget exchange information about messages in two device registers, SendRegister and ReceiveRegister, which are structured data areas imported by the nugget. A message consists of a header composed by the nugget and a body composed by the client. The body may be divided into several (up to MAXMESSPARTS) disjoint regions, each contiguous in kernel virtual space. A message body is described by the BufferDescrip field of the SendRegister or ReceiveRegister. BufferDescrip is an array describing all the parts of a client's message's body. Each array entry describes one region by a start address and length. A length of 0 indicates a missing region. The total length of all regions must lie between MINMESSLENGTH and MAXMESSLENGTH.

10.3. Clock

The nugget provides a timing service called the *virtual clock*. This service is continuous and not invoked by a procedure call. The virtual clock consists of a data area supplied by the client. It contains a field called Count, which the client may set and inspect at will. Every tick of the virtual clock (currently, 60th of a second) causes Count to be decremented. Count may become negative. If the count becomes 0, then the client is interrupted at CPU priority CLOCKPRIORITY.

10.4. Disk

Virtual disks are created and destroyed by users interacting with the nuggetmaster. A user who builds a partition may specify that certain virtual disks be made part of that partition. Each virtual disk is emulated by some physical disk, attached to some physical node. One physical disk may emulate many virtual disks. If the physical node associated with a virtual disk is part of a user's partition, then no other users have access to virtual disks on the same physical disk. This association allows the client to make reliable timing measurements of disk operations. Alternatively, the virtual disk's node may be omitted from the partition. In this case, the clients in the partition still have access to the virtual disk, but other partitions may use the same physical disk at the same time. We call the node connected to a shared physical disk outside of any partition a *diskserver*.

The client can only access a virtual disk through the nugget, so virtual disks that are not in the partition are protected from tampering. The nugget on each node in a partition has access to all virtual disks in the partition. The user may of course decide to submit virtual disk access requests from only one client.

Each virtual disk is assigned a number in the range 0 to No_of_Disks, a variable imported and initialized by the nugget. Cylinders on the disk are referred to by logical cylinder numbers. Each virtual disk has a set of registers declared in the definition file and imported by the nugget. These registers provide communication between the client and the nugget.

The nugget uses the registers to describe the parameters of the disk, including information on number of cylinders, surfaces per cylinder, and sectors per surface. If the physical disk is connected to this node, then the nugget also provides information on the current position of the disk arm.

The client uses the registers to specify the type of operation ("read" or "write"), the logical cylinder, the surface, the sector, the number of sectors, and the kernel virtual address of the buffer to be used in the operation. Operations are started by calling the procedure DiskOp, passing as a parameter the name of the virtual disk involved in the operation. The client is informed by an interrupt when the operation has completed. In order to complete the operation, the nugget may have to exchange messages with the nugget on the node that is connected to the associated physical disk.

10.5. Terminals

The user may request that each machine be provided with a number of *virtual terminals* to be multiplexed to the user's terminal on the host. The user can then interact with one physical terminal and view or control the progress of the entire partition. Control of this "multi-terminal" is provided by commands to the nuggetmaster, which are detailed in the nuggetmaster chapter.

The number of virtual terminals on each node is known to the client in the variable No_of_Terminals, imported and initialized by the nugget. Each terminal is referred to by a number in the range 0 to No_of_Terminals-1. Each node has at least one virtual terminal, the *console*, numbered 0. Each terminal has its own set of registers. All registers are exported by the client in the array TermDevices.

To write to a terminal, the client places the character to be written in Output_Char of the terminal's register and calls OutputReady, passing the name of the virtual terminal. When the terminal can accept new output (the last character has been sent to the nuggetmaster) the client is notified by interrupt. The nuggetmaster may buffer the character until it can be displayed. To read from the terminal, the client calls EnableInput, passing the name of the virtual terminal. The client is notified by interrupt when a character arrives from the nuggetmaster.

10.6. Debugger

Lastly, the nugget provides a simple synchronization and debugging service to the client:

```
   procedure Pause; external;
```

A client that invokes Pause does not continue until the pause is released by the nuggetmaster. No interrupts of any kind are seen by the client during pause. The nugget remains active during the pause state and continues to send and receive messages. Notification of successful operations is given after the pause state ends. Of course, the client is unable to submit new requests while pausing. The client's virtual clock is not decremented in pause state. During pause, many debugging and partition-control commands can be executed through the nuggetmaster and are outlined in a later chapter.

The nugget also provides a simple means for debugging itself:

```
const
            NUGSTATESIZE = 500;
type
            NuggetState = array 0:NUGSTATESIZE-1 of shortint;
procedure Save_Nugget_State(var SavedState : NuggetState);
            external;
```

The procedure Save_Nugget_State raises priority to that of the clock and copies all data used by the nugget to SavedState. These data include the state of the communication device. The saved state may be inspected to determine unexplained actions.

11. Communication Interface

The nugget controls all communications among clients residing in the nodes of a partition. Communication is by messages. The communication service appears to the client as a device that sends and receives messages. Relevant declarations can be found in this chapter and in the chapter detailing the Modula definition file at the end of this document.

The client can send and receive messages by calling NuggetSend and NuggetReceive. The client implicitly passes parameters describing the operation in the device registers SendDevice and ReceiveDevice. These devices are part of CommDevice, a data area exported by the client and imported by the nugget. When a communication operation completes, the client is notified by interrupt at CPU priority level DEVICEPRIORITY through SendIntVect and ReceiveIntVect. These two vectors should be initialized by the client before lowering CPU priority in the client's initialization process.

The client can choose between two modes for message delivery. "Datagram" mode writes the message out on the line. The nugget does not guarantee its delivery but will make a good faith effort to get it to the nugget of the other machine. "Ack-receipt" mode is a reliable delivery service. The nugget will not only deliver the message to the other nugget, but will wait for notification by the other nugget that its client accepted the message. The client is informed if delivery to the other client is not possible.

The client refers to other nodes through logical node addresses. These must be in the interval NUGGETMASTER to MaxLogicalAddr: NUGGETMASTER (constant 0) is the logical address of the nuggetmaster, and MaxLogicalAddr is the number of the nodes in the partition. The client's own machine's logical name can be found in SelfLogicalAddr. These two variables are supplied by the client and initialized by the nugget.

A message consists of a header composed by the nugget and a body composed by the client. The body may be divided into several (up to MAXNOMESSPARTS) disjoint regions, each contiguous in kernel virtual space. A message body is described by the BufferDescrip field of the SendRegister or ReceiveRegister. BufferDescrip is of type MessBodyDescrip, which is an array describing all the parts of a client's message's body. Each array entry describes one region by the address of the start of the region (BodyAddr) and the length in bytes of the region (Length). A length of 0 means that the BodyAddr field is not valid, and that this region is missing. Length must be an even number of bytes. The sum of all lengths must lie between MINMESSLENGTH and MAXMESSLENGTH.

The SendRegister fields are interpreted as follows:

```
------------------------------------------------------------------
type
        InterruptVector =
                record
                        InterruptHandler : KernelAddr;
                        NewPSW : midint;
                end; (* InterruptVector *)
        PartDescrip =
                record (* describes one part of a message body *)
                        BodyAddr : KernelAddr;
                        Length   : longint;
                end; (* PartDescrip *)
        MessBodyDescrip = array 0:MAXMESSPARTS-1 of PartDescrip;

var
        CommIntVectors :
                record  (* the communication interrupt vectors *)
                        SendIntVect : InterruptVector;
                                (* client initializes to interrupt handler's
                                address and desired lower half of PSW upon
                                interrupt *)
                        ReceiveIntVect : InterruptVector;
                                (* not used in sending *)
                end; (* CommIntVectors *)
        SendRegister :
                record (* client sets all fields *)
                        BufferDescrip : BodyDescrip;
                                (* points to each part of message *)
                        Mode : (DataGram,AckReceipt);
                                (* desired communication circuit *)
                        DestAddr : LogicalNodeAddr;
                                (* destination's logical node address *)
                end; (* SendRegister *)

procedure NuggetSend : StandardInt; (* sends SendRegister's message *)
------------------------------------------------------------------
```

To send a message, the client first sets the fields of SendRegister. BufferDescrip should point to the message buffer, with each array entry's kernel virtual address (BodyAddr) and length in bytes (Length) corresponding to the parts of the message buffer. Mode and destination address (DestAddr) must also be set. The client then calls NuggetSend.

NuggetSend returns the standard legal/illegal flag. A call to NuggetSend is illegal if a pending send to the same destination node has not yet been dispatched and acknowledged. The client may have MaxLogicalAddr sends pending. The client may send to the nuggetmaster, logical machine 0, but not to itself.

When the send completes the client will be interrupted at CommIntVectors.SendIntVect.InterruptHandler with the lower 16 bits of the PSW as set in CommIntVectors.SendIntVect.NewPSW. These interrupt fields are typically set by the client at initialization time. Changes to these fields during a send operation will produce unpredictable results.

If client the Mode is AckReceipt, the send cannot complete successfully unless the message is acknowledged. Completion may mean that the nugget timed out trying to send and that the message was not received. At completion of the send the nugget pushes onto the stack either 0 or 1 to indicate that the send operation succeeded (0) or timed out (1) and then pushes the destination's logical node address. These two parameters should be popped from the stack before the client returns from the interrupt. There is no ready bit or interrupt-enable bit in SendRegister; client-notification interrupts are always enabled.

In addition to sending a message to a single remote machine, the client may send a single broadcast message to all nodes in the partition. The nugget does not deliver broadcast messages to the sending client. A broadcast message must be a datagram. In addition to one send directed to every other node in the partition, the client may have one broadcast request active. To send a broadcast message the client specifies BROADCAST (value -1) as the node address when requesting a send. Broadcast messages are received as if they were a node-to-node datagram.

The fields in ReceiveRegister are interpreted as follows:

```
--------------------------------------------------------------------
var
        CommIntVectors :
                record  (* communication interrupt vectors *)
                        SendIntVect : InterruptVector; (* not used in receive *)
                        ReceiveIntVect : InterruptVector;
                                (* client sets to virtual address of interrupt
                                handler, and desired lower 16 bits of PSW upon
                                interrupt *)
                end; (* CommIntVectors *)
        ReceiveRegister :
                record
                        (* the following are set by the client *)
                        BufferDescrip : BodyDescrip;
                                (* client sets to point to all parts of buffer *)
                        (* the following are set by the nugget *)
                        Status : (MessArrived, NoMessTimeOut);
                                (* type of interrupt *)
                        SourceAddr : LogicalNodeAddr;
                                (* source's logical node address *)
                        Mode : (DataGram,AckReceipt);
                                (* communication circuit of message *)
                        Length : longint;
                                (* total length of arrived message *)
                end; (* ReceiveRegister *)

procedure NuggetReceive : StandardInt;
        (* enables receipt of message *)
--------------------------------------------------------------------
```

To receive a message, the client sets the BufferDescrip field of ReceiveRegister to point to the message buffer by setting each array entry's virtual address (BodyAddr) and length in bytes (Length) to those values corresponding to the parts of the message buffer. The buffer should be large enough to hold any expected message. Each part will be filled to capacity by the incoming message before the next part is used. Messages that do not fit into the buffer will not be accepted or acknowledged. The sum of all Lengths must be at least MINMESSLENGTH. The receiving client does not specify which client it wishes to receive from.

NuggetReceive returns the standard legal/illegal flag. A call to NuggetReceive is illegal if a pending receive has not yet been fulfilled. Several sends and one receive may be outstanding at the same time.

When a message is accepted by the nugget from another client, the client will be interrupted at CommDevice.ReceiveIntVect.InterruptHandler with the lower 16 bits of PSW as set in CommDevice.ReceiveIntVect.NewPSW. These interrupt fields are typically set by the client at initialization time. Changes to these fields during a receive operation will produce unpredictable results.

The client's receive request may be interrupted by a timeout that indicates that no messages from any other node has arrived. The nugget sets ReceiveRegister.Status to distinguish between the receipt of a message (MessArrived) and the timeout (NoMessTimeOut). If Status is MessArrived, SourceAddr will be set to the source's logical node address, Mode will be set to the mode of the newly arrived message, and Length will be set to the total length of the message. The priority level of the processor upon interrupting the client will be the priority of the device (DEVICEPRIORITY). There is no ready bit or interrupt-enable bit in ReceiveRegister. Client notification interrupts are always enabled.

The entire communications definition file is reproduced at the end of this document.

12. Virtual Clock Interface

The declarations for the virtual clock are as follows:

```
-----------------------------------------------------------------------
const
        MIN_COUNT = -32767; (* Count will not fall below this value *)
        CLOCKPRIORITY = 24; (* clock priority level *)

var
        TimeIntVector : InterruptVector;
        TimeDevice =
                record
                        Count : longint;
                end; (* TimeDevice *)
-----------------------------------------------------------------------
```

The client may set or inspect the Count field at will. Every tick (currently, 60th of a second), the nugget decrements Count. The count may become negative. The nugget will not decrement Count if so doing would change it from negative to positive. Count will therefore not fall below MIN_COUNT. If the count becomes 0, then the client is interrupted at location TimeIntVector.InterruptHandler with the lower 16 bits of the PSW set to TimeIntVector.NewPSW. These fields should be set by the client at initialization time. Changes to these fields when the client clock is positive will produce unpredictable results. The priority level will be that of the clock (CLOCKPRIORITY).

The client may change Count at any time. To change the interrupt vector, the client should first set Count to any negative integer, disabling the clock interrupt. Count is initialized by the nugget to MIN_COUNT.

13. Virtual Disk Interface

The virtual disk program presents a disk device to the client. The program controls access to the disk, thereby protecting the client's data from access by other partitions over time and preventing the client from accessing disk data belonging to other partitions. Virtual disks are created by users interacting with the nuggetmaster. A user may include access to one or more virtual disks in a partition. Each node in the partition may access those virtual disks. Virtual disk names are uniform across the partition. The definition file contains a section defining the virtual disk interface. This part of the include file can be found at the end of this chapter.

The client refers to each virtual disk by its logical disk number. The maximum such reference is in No_of_Disks, which is imported and initialized by the nugget. All cylinder references are logical numbers as well. Surface and sector numbers are physical.

Each virtual disk has its own set of device registers. These are accessed through the array DiskDevice. Each register has three sets of fields. First, the static fields, initialized by the nugget, describe the device (Max_Logical_Cylinder, Num_Surfaces, Num_Sectors_Per_Track, and SectorLength) and should not be written by the client.

Second, the dynamic fields describe the position of the disk arm (Current_Cylinder) and the status of the I/O operation (Status_of_I_O). These fields are updated by the nugget during I/O operations and should not be written by the client. If the associated physical disk is not attached to this node, Current_Cylinder contains the constant NOTDEFINED. Status_of_I_O contains the status of the last completed I/O operation. It is reset with each call to the procedure DiskOp. The status can indicate two failure modes: An actual disk operation failed (DiskFailure), and there was a possible failure of the I/O operation (MessFailure). In the second case, communication with a remote disk failed; the I/O operation may have succeeded.

Finally, command fields describe the I/O operations. This information is written by the client before calling DiskOp. "Operation" indicates which operation is requested. "Cylinder", "Surface" and "Sector" specify the starting position of the I/O. A seek operation to that track is implied. "Sector_Count" specifies the number of contiguous sectors to be transferred. Sector_Count must not exceed Num_Sectors_Per_Track. "BufferAddr" holds the starting address of the buffer used for the transfer. The buffer must be virtually contiguous in kernel space.

To schedule a disk operation, the client sets all the fields of RequestInfo and then calls DiskOp, passing the logical disk number of the virtual disk. DiskOp returns the usual legality indication. DiskOp can fail if the disk name is out of range, Request_Info is illegal, or the previous disk operation has not yet completed. A legal call will schedule the disk operation.

Upon completion of the disk operation, the client is interrupted at the address stored in DiskIntVector. The virtual disk number is pushed onto the stack as a longint prior to the interrupt. This parameter should be removed before the return from interrupt. The interrupt routine will run at the priority level of the virtual disk (DEVICEPRIORITY). This priority level is the same as the communication device. The lower 16 bits of the processor status word is set to NewPSW of the interrupt vector prior to the interrupt. It is the client's responsibility to save the machine state upon interrupt.

The disk interface section of the definition file follows:

```
-----------------------------------------------------------------------------
const
        MAXDISKS = ? (* max number of virtual disks per machine *)
        NOTDEFINED = -1; (* position information not available *)

type
        OpInfo =  (* command fields of a device register *)
                record
                        (* these fields are client read/write *)
                        Operation : (Read_Disk, Write_Disk);
                        Cylinder : longint; (* logical cylinder (track) *)
                        Surface  : longint; (* physical surface *)
                        Sector   : longint; (* physical sector *)
                        Sector_Count : longint; (* number of sectors to
                                transfer *)
                        BufferAddr : KernelAddr; (* kernel virtual address
                                of start of data *)
                end; (* OpInfo *)
        DiskRegisters =
                record
                (* static fields, initialized by nugget, read by client *)
                        Max_Logical_Cylinder : longint;
                        Num_Surfaces : longint;
                        Num_Sectors_Per_Track : longint;
                        SectorLength : longint; (* number of bytes per disk
                                sector *)
                (* dynamic fields, set by nugget, read by client *)
                        Current_Cylinder : longint; (* current position of
                                disk arm or NOTDEFINED *)
                        Status_of_I_O : (Success, DiskFailure, MessFailure);
                (* command fields, set by client, read by nugget *)
                        RequestInfo : OpInfo;
                end; (*DiskRegisters*)

var
        No_of_Disks : longint; (* actual number of virtual disks,
                bounded by MAXDISKS, initialized by nugget *)
        DiskIntVectors : array 0:MAXDISKS-1 of InterruptVector;
        DiskDevice : array 0:MAXDISKS-1 of DiskRegisters;

procedure DiskOp(DiskNo : longint) : StandardInt; external;
        (* schedules a virtual disk operation *)
```

14. Virtual Terminal Interface

Virtual terminal service allows each client in a partition to access a terminal (for both input and output) that is multiplexed to the user's terminal on the host. The definition file contains a description of the virtual terminal interface. This interface can be found at the end of this chapter.

The virtual terminal "device" is controlled by nugget routines. Many of these routines communicate with the nuggetmaster's virtual terminal handler. The nuggetmaster specification details the commands that the user may employ to control the multiplexed physical terminal.

When a partition is specified by the user, each node is given one or more virtual terminals logically numbered 0 to No_of_Terminals-1. The variable No_of_Terminals is imported and initialized by the nugget. Terminal 0 is always present and acts as the node's console.

The virtual terminal appears to the client as a device. The client must initialize the device by setting the interrupt vectors. When a character has appeared on the input device, the client is notified by interrupt at InputIntVect.InterruptHandler at the priority of the terminal (DEVICEPRIORITY) providing that the client has called EnableInput on this device. The terminal priority is the same as that of the communication virtual device. Before calling the client's interrupt handler, the nugget pushes the virtual terminal number on the stack as a longint. This parameter should be removed by the client before returning from the interrupt.

When a character has been written by the output device, the client is notified at OutputIntVect.InterruptHandler. Again, the virtual terminal number is pushed on the stack prior to the call. For both interrupts, the lower 16 bits of the processor status word are set to NewPSW of the interrupt vector.

To read a character, the client calls EnableInput, passing the terminal number as a longint parameter. EnableInput returns a legality indicator. The Input_Status of the terminal is Device_Busy until the character arrives. After the interrupt notification, the client can read Input_Char. The client can examine Input_Status to see if the input character has been lost due to device errors.

To write a character, the client should set write the character to Output_Char of the appropriate entry in TermRegs, and call OutputReady, passing it, as a longint parameter, the virtual terminal number. OutputReady returns the usual legality indicator. After the character is written to the device (that is, sent to the nuggetmaster), the Output_Status of this device is set, the virtual terminal number is pushed on the stack, and the client is notified through its interrupt handler. This parameter should be removed before returning from interrupt. If the line to the nuggetmaster goes down, a value of DeviceError appears in the status field.

Flow control between the nugget and the physical terminal on the host is managed by a permission scheme whereby each end sends permissions to the other for characters. If the client tries to write to the virtual terminal, but the nugget does not have permission to send the character to the nuggetmaster, the output-completion interrupt is delayed until the character can be sent. If the user enters characters on the physical terminal that the nuggetmaster does not have permission to send, some characters are buffered. If the buffer is exceeded, no more characters are accepted. It is useful for the client to echo input so the user can tell how much has been accepted. See the nuggetmaster specifications for details.

The client/virtual terminal interface follows:

```
-------------------------------------------------------------------------------
const
        MAXTERMINALS = ? (* maximum number of terminals per machine *)

type
        Status_Values = (DeviceBusy, CharOkay, DeviceError)
        TerminalRegisters =
                record
                        (* the following are client read only *)
                        Input_Char : char;  (* terminal input character *)
                        Input_Status : Status_Values; (* stat input stream *)
                        Output_Status : Status_Values; (* stat output stream *)
                        (* the following are client write only *)
                        Output_Char : char;  (* terminal output character *)
                end; (* TerminalRegisters *)

var
        TermIntVectors :
                record
                        InputIntVect  : InterruptVector;
                        OutputIntVect : InterruptVector;
                end; (* TermIntVectors *)
        No_of_Terminals : longint; (* actual number of terminals on node *)
        TermDevices : array 0:MAXTERMINALS of TerminalRegisters;

procedure EnableInput(TerminalNumber : longint) : StandardInt;
        (* client call to enable next input from TerminalNumber *)
        external;

procedure OutputReady(TerminalNumber : longint) : StandardInt;
        (* client call to write Output_Char of
                TermRegs[TerminalNumber] *)
        external;
-------------------------------------------------------------------------------
```

15. Node Operation and Control

The node may be in one of several states. The nuggetmaster can cause the node to change states by sending commands to the nugget. The client can also cause the node to change states. The nugget sends a status message to the nuggetmaster upon state change. The set of states is as follows:

1. BOOT: A bootstrap program (very likely implemented on the boot tape) waits for a load image from any nuggetmaster. That image contains a nugget bound to a client. The nugget's tables already indicate the nature of

the partition. The bootstrap program transfers control to a fixed location in the nugget. The nugget, after initializing itself, transfers control to the client (at a fixed symbolic location: _client) and enters RUNNING state.

2. HALT: An actual 'halt' instruction has been executed. The node must be rebooted to do anything. In this state, interrupts are ignored. The purpose of this state is for failure insertion (and completeness, since the client may execute a halt instruction).

3. PAUSE: The user or client has requested a pause. The nugget masks all interrupts to the client and busy-waits on the communication device for a CONTINUE order from the nuggetmaster. The client's clock is frozen (that is, not decremented). The nugget remains active and continues to process outstanding client send and receive requests as well as outstanding virtual disk requests, but delays notifying the client about any completing operation until the CONTINUE order arrives. The nugget does not accept input characters from virtual terminals during the PAUSE state. When the CONTINUE order arrives, the nugget and client resume RUNNING state.

4. RUNNING: After a successful boot, the nugget and client are running normally.

The nuggetmaster provides the user several commands that modify the node's state as well as commands for simple debugging. The user's commands are implemented through messages from the nuggetmaster to the nugget. The commands that the nuggetmaster may give the nugget follow:

1. REBOOT: Returns node to the boot state. This order is legal in any state.

2. RESTART: Returns the client to the running state at symbolic location _client. This order is legal in RUNNING or PAUSE states. The order is obeyed following completion of all outstanding client requests. During this delay, the node runs in the PAUSE state. Users who intend to restart their

clients should take pains to make them re-entrant. In particular, initialized data must be re-initialized on every entry to _client.

3. HALT: Puts the node in the halt state. This order is legal in RUNNING or PAUSE states. Outstanding client requests are not honored by the nugget.

4. HELLO: This order is ignored by the nugget, but since every message from the host is acknowledged, it can be used to check if the nugget is still alive. This order is legal in RUNNING or PAUSE states.

5. PEEK: The order includes a 32-bit physical address. The nugget returns the 16 bits contained in the word at that address to the nuggetmaster. This order is legal in RUNNING or PAUSE states.

6. POKE: The order includes a 32-bit physical address and a 16-bit value to store. The nugget stores that value in the word at that location. This order is legal in RUNNING or PAUSE states.

7. CONTINUE: Moves the nugget from the pause state to the running state. This order is legal in the PAUSE state only. It is ignored in the RUNNING state. The client is notified of any requests that completed in the PAUSE state.

8. PAUSE: Puts the nugget and client in the pause state. This order is legal in the RUNNING state only. This command has the same effect as the client calling the procedure Pause.

16. Nugget's Communication Protocol

The nugget is in charge of all messages sent between its node and other nodes or the host. The nugget places a header on each message it sends. This header is described by the type Header defined below. The first group of fields in Header are used in the inter-machine protocol. The rest of the fields are used for internal multiplexing of messages on the node machine or by the nuggetmaster on the user's host.

The following declarations are not part of the definition file:

```
-----------------------------------------------------------------------
const
        HEADSIZE = 10; (*in bytes*)

type
        NodeAddr = shortint;

const
        BROADCAST = NodeAddr(-1);   (* as a destination *)
        NUGGETMASTER = NodeAddr(0); (* logical node address *)

type
        Header =
                record
                (* inter-machine protocol fields *)
                        (* hardware-dependent fields *)
                                Dest : NodeAddr;
                                Source        : NodeAddr;
                        (* software dependent fields *)
                                Version   : shortint;  (* For backward
                                        compatibility *)
                                BitAccess : shortint;  (* AckNo,SeqNo,Datagram,
                                        BareAck,Rsrvds *)
                                Reserved  : midint;  (* For later use *)
                                Length        : integer;  (* The length of the
                                        message body in bytes *)
                (* local addressing fields *)
                        Dest_Socket_No, Source_Socket_No : shortint;
                end; (* Header *)
-----------------------------------------------------------------------
```

The inter-machine protocol fields are divided into hardware-dependent and software protocol fields. Following the hardware-dependent fields is a version number (Version). Different versions of the protocol may exist on the network concurrently without interfering with each other.

Local addressing fields represent socket numbers. There is a separate socket number for each of four purposes: client messages, virtual terminal messages, virtual disk messages, and command messages. The socket numbers on nodes are bound at design time. The socket numbers on the host are determined when the partition is built, and refer to Unix ports.

The nugget uses a one-bit stop-and-wait protocol to send messages. This protocol uses the destination address (Dest), source address (Source), the sequence number of the message (bit 7 of BitAccess), and the sequence number of the next message expected (otherwise known as the acknowledgement bit, bit 8 of BitAccess). The other bits describe additional variations.

Bit 6 of BitAccess, if set, indicates that an acknowledgement is not expected; the message is a datagram. Datagram messages have higher priority than ack-receipt messages and advance to the head of the queue. Datagram messages do not carry acknowledgements for previously received messages. The sequence number of the datagram message is ignored by the receiver and does not reset any Delta-T timers described below.

Bit 5 of BitAccess, if set, indicates that this message is a bare acknowledgement. Bare acknowledgements only use fields through BitAccess, hence are only four bytes long. A bare acknowledgement is not a datagram and must have bit 6 clear.

The other 4 bits of BitAccess are reserved for future use. "Reserved" is also reserved for future use.

Length is the length in bytes of the rest of the message, including fields of the header that follow Length.

The two fields, Dest_Socket_No and Source_Socket_No, are used to multiplex messages among the client, nugget or nuggetmaster, virtual terminal, and virtual disk.

The nugget uses the delta-T protocol to recover from lost messages, dead machines, or dead lines. This protocol uses three timing constants: MPL (Maximum Packet Lifetime), A (maximum Acknowledgement delay), and R (maximum Retransmission time). The nugget stops retransmitting an unacknowledged message after R ticks of the clock and announces failure to the client. The minimum wait between retransmissions is 2MPL+A.

State information for a communication channel consists of the SeqNo and the AckNo bits sent with messages on that channel. If a message is acknowledged, the sender's state is changed by flipping the SeqNo bit and the receiver's state is changed by flipping the AckNo bit. The nugget maintains state information for 2MPL+A+R ticks upon receipt of any message. If no message arrives during this interval, the receiver resets the protocol: The next message that arrives defines the state of the sender and is accepted. A sender will try to retransmit unacknowledged messages for R ticks. If the message is still unacknowledged, all sending to that destination is blocked for 3MPL+A+R ticks. The receiver will then reset its state and accept the next message. When the nugget initializes, it waits at least 3MPL+A+R ticks before sending or receiving messages. Any previous conversations are thereby flushed from the network. Datagram messages are ignored for the purpose of this protocol, that is, such messages do not cause any timers to be reset.

17. The definition file

The Modula "include" file in /usr/crystal/nugget/MC_include documents the common structures used by the client and the nugget. The following is the entire definition file.

```
----------------------------------------------------------------------
module nugget;

define (* interface routines *)
        NuggetSend, NuggetReceive, (* communication routines *)
        DiskOp, Get_Disk_Info,        (* virtual disk routines *)
        EnableInput, OutputReady,  (* virtual terminal routines *)
        Pause, Save_Nugget_State;  (* debug routines *)

use (* parameters *)
        CommDevice, CommIntVectors, (* communication *)
        TimeDevice, TimeIntVector,  (* virtual clock *)
        _client,        (* client entry point after nugget initialization *)
        MaxLogicalAddr, SelfLogicalAddr, (* partition data *)
        No_of_Disks, DiskDevices, DiskIntVectors, (* virtual disk *)
        No_of_Terms, TermDevices, TermIntVectors; (* virtual terminal *)

(* initialization declarations *)

procedure _client; external; (* the client's entry point *)

(* communication declarations *)

const
        MINMESSLENGTH = 0; (* max size (in bytes) of client messages *)
        MAXMESSLENGTH = 2000;
        MAXMESSPARTS = 3;  (* max number of parts of message body *)
        DEVICEPRIORITY = 20; (* device priority level *)
        BROADCAST = -1;  (* when used as a destination address *)
        NUGGETMASTER = 0;        (* logical address of nugget master *)

type
        LogicalNodeAddr = shortint;
        KernelAddr = integer; (* virtual address in kernel space *)
        PhysicalAddr = longint;  (* physical machine address *)
        InterruptVector =
                record
                        InterruptHandler : KernelAddr; (* in kernel space *)
                        NewPSW : midint; (* desired lower 16 bits of PSW *)
                end; (* InterruptVector *)
        PartDescrip = (* describes one part of client's message *)
                record
                        BodyAddr : KernelAddr;  (* start of part *)
                        Length : longint;  (* size of part in bytes *)
                end; (* BodyDescrip *)
        MessBodyDescrip = array 0:MAXMESSPARTS-1 of PartDescrip;

var
        CommDevice :
                record  (* communication send and receive registers *)
                        SendRegister :
                                record
                                        (* the following fields are read/write *)
                                BufferDescrip : MessBodyDescrip;
                                Mode : (DataGram,AckReceipt);
                                DestAddr : LogicalNodeAddr;
                        end; (* SendRegister *)
                ReceiveRegister :
                        record
                                (* the following are client read/write *)
                                BufferDescrip : MessBodyDescrip;
                                (* the following are client read only *)
                                Status : (MessArrived, NoMessTimeOut);
                                SourceAddr : LogicalNodeAddr;
                                Mode : (DataGram,AckReceipt);
                                Length : longint;
                        end; (* ReceiveRegister *)
        end; (* CommDevice *)
    CommIntVectors :
        record  (* communication device interrupt vectors *)
                (* the following are client read/write *)
                SendIntVect  : InterruptVector;
                ReceiveIntVect : InterruptVector;
        end; (* CommIntVectors *)
    MaxLogicalAddr, SelfLogicalAddr : NodeAddr; (* partition data *)

procedure NuggetSend : integer; external;
    (* enables sending of the message described in SendRegister *)
procedure NuggetReceive : integer; external;
    (* enables receipt of messages described in ReceiveRegister *)

(* debugging features *)

procedure Pause; external;

const
    NUGSTATESIZE = 500;
type
    NuggetState = array 0:NUGSTATESIZE-1 of shortint;

procedure Save_Nugget_State(var SavedState : NuggetState); external;

(* clock declarations *)

const
    MIN_COUNT = -32767; (* Count will not fall below this value *)
    CLOCKPRIORITY = 24; (* clock priority level *)

var
    TimeIntVector : InterruptVector;
    TimeDevice =
        record
                Count : longint;
        end; (* TimeDevice *)

(* virtual disk declarations *)
```

```
const
        MAXDISKS = ? (* max number of virtual disks per machine *)
        NOTDEFINED = -1; (* position information not available *)

type
        OpInfo =  (* command fields of a device register *)
                record
                        (* these fields are client read/write *)
                        Operation : (Read_Disk, Write_Disk);
                        Cylinder : longint; (* logical cylinder (track) *)
                        Surface  : longint; (* physical surface *)
                        Sector   : longint; (* physical sector *)
                        Sector_Count : longint; (* number of sectors to
                                transfer *)
                        BufferAddr : KernelAddr; (* kernel virtual address
                                of start of data *)
                end; (* OpInfo *)
        DiskRegisters =
                record
                (* static fields, initialized by nugget, read by client *)
                        Max_Logical_Cylinder : longint;
                        Num_Surfaces : longint;
                        Num_Sectors_Per_Track : longint;
                        SectorLength : longint; (* number of bytes per disk
                                sector *)
                (* dynamic fields, set by nugget, read by client *)
                        Current_Cylinder : longint; (* current position of
                                disk arm or NOTDEFINED *)
                        Status_of_I_O : (Success, DiskFailure, MessFailure);
                (* command fields, set by client, read by nugget *)
                        RequestInfo : OpInfo;
                end; (*DiskRegisters*)

var
        No_of_Disks : longint; (* actual number of virtual disks,
                bounded by MAXDISKS, initialized by nugget *)
        DiskIntVectors : array 0:MAXDISKS-1 of InterruptVector;
        DiskDevice : array 0:MAXDISKS-1 of DiskRegisters;

procedure DiskOp(DiskNo : longint) : StandardInt; external;
        (* schedules a virtual disk operation *)

(* virtual terminal declarations *)

const
        MAXTERMINALS = ? (* maximum number of terminals per machine *)

type
        Status_Values = (DeviceBusy, CharOkay, DeviceError)
        TerminalRegisters =
                record
                        (* the following are client read only *)
                        Input_Char : char;  (* terminal input character *)
                Input_Status : Status_Values; (* stat input stream *)
                Output_Status : Status_Values; (* stat output stream *)
                (* the following are client write only *)
                Output_Char : char;  (* terminal output character *)
            end; (* TerminalRegisters *)

var
    TermIntVectors :
            record
                InputIntVect  : InterruptVector;
                OutputIntVect : InterruptVector;
            end; (* TermIntVectors *)
    No_of_Terminals : longint; (* actual number of terminals on node *)
    TermDevices : array 0:MAXTERMINALS of TerminalRegisters;

procedure EnableInput(TerminalNumber : longint) : StandardInt;
    (* client call to enable next input from TerminalNumber *)
    external;

procedure OutputReady(TerminalNumber : longint) : StandardInt;
    (* client call to write Output_Char of
            TermRegs[TerminalNumber] *)
    external;

end nugget;
-----------------------------------------------------------------
```