Bell Laboratories

Subject: **Datakit-CMC Protocols and Procedures**
Case- 11173 -- File- 39199-11

date: **April 14, 1982**

from: **G. L. Chesson**

TM: **82-11273-1**

MEMORANDUM FOR FILE

1.0 Introduction

Numerous experimental protocols and software techniques have been developed in the last few years on A. G. Fraser's experimental Datakit hardware at Murray Hill. Most of this document is devoted to a detailed exposition of some of the protocols and interfaces that were implemented. Section 2 describes the device driver and low-level operating system interfaces. Sections 3.0 through 3.4 describe the low-level addressing, connection management, and maintenance facilities. Sections 3.5 and 3.6 discuss higher-level addressing and control mechanisms. Section 4 describes the message interfaces to the basic network servers that were built. Section 5 describes the experimental trailer protocol for data transport. Section 6 describes the software library of network access routines that implement the section 3 protocols. A permuted index of technical terms referenced to page numbers is included at the end. The remainder of this section will present underlying design concepts.

1.1 Background Information

Datakit architecture was first described in the open literature by Chesson [1] and Fraser [2] and more recently by Luderer, Che, and Marshall [3] and will not be considered in depth here. For this discussion it suffices to say that the system utilizes a packet-switching concept implemented in hardware. A Datakit *switch* is a single circuit board in a chassis with printed circuit backplane called a *node*. Other circuit boards plug into a node and perform functions such as interfacing to computers, terminals, and other nodes. All circuit boards that work in the Datakit node environment are called *modules*.

Each module may access up to 511 logical *channels* for transmitting and receiving on the backplane. All data travels first from modules to the switch. The switch contains a control mechanism for routing this data based on channel number and module number of the sender; i.e. data coming into the switch from any module/channel can be routed to any other module/channel in the node. The translation from the sender's module/channel number to a destination module/channel is controlled by a routing table in the control memory of the switch. Note that the switch merely operates as directed by the control memory. The memory contents are the responsibility of network control software that sets up the routing tables in response to requests for service, i.e. data paths through a switch.

[1] G. L. Chesson, "Datakit Software Architecture", *Proc. ICC 79*, June 1979, Boston Ma., pp.20.2.1-20.2.5.
[2] A. G. Fraser, "Datakit - A Modular Network for Synchronous and Asynchronous Traffic", *Proc. ICC 1979*, June 1979, Boston, Ma.,pp.20.1.1-20.1.3.
[3] G. W. R. Luderer, H. Che, W. T. Marshall, "A Virtual Circuit Switch as the Basis for Distributed Systems", *Proc. Seventh Data Communications Symposium*, October 1981, Mexico City, Mexico, pp.164-178.

The architecture is not limited to a single switch: *trunk* modules provide data paths between different switches. This in turn provides the basis for network growth and adaptability. There is a strong analogy between this architecture and the telephone system in that a Datakit node and its related equipment corresponds to a telephone central office and a Datakit trunk corresponds to an inter-office carrier system.

1.2 Network Access

As explained above, Datakit provides virtual circuit service at a primitive level as distinguished from other network architectures that build virtual circuit service on top of a datagram layer. The consequences of this design are that network nodes must be centrally controlled by ESS-like software programs and a network user needs a virtual circuit to access any network service. It should come as no great surprise that a fair amount of effort goes into the design of virtual circuit control mechanisms for a network like Datakit.

There are at least two views of network access: that from the user or host computer software where the network is used, and that of the network control software providing service. These may be at odds with one another. In some designs it may be that keeping one part simple complicates the other. The design of the software described here favors simplicity at the host software interface, partly on the grounds that the best way to proliferate a network is to make it understandable and partly in reaction to the overwhelming nature of conventional network access methods.

The user interface for network access is constrained to a single *request/response* message handshake for all network services. The basis of the handshake is a message format called a *dialout* structure that defines a uniform syntax for all network control messages. The same protocol model is used throughout: send a *dialout* structure to the network control software and wait for a response. A prototype routine for obtaining a virtual circuit is displayed in section 3.8, and the interfaces to some network access library routines are documented in section 6.

The *dialout* structure defined in section 3 is actually the result of four major iterations in software. It may be of interest to note that one iteration was for a fundamental algorithm change in setting up channels, another was for improvements in the logical addressing scheme, and the others were solely to increase the message size.

1.3 Switching Control

Switching as a generic term in telephony refers to the mechanics of causing two telephones to talk to one another. In a network like Datakit it refers to the mechanics of providing virtual circuits between network addresses. The switching control documented in the rest of this paper centers around a software process called the CMC, or *common control* program. The organization of the CMC and related software borrows from the principles of modularity demonstrated by the hardware design.

The Datakit hardware has the property that logically separate operations or functions are implemented by physically separate hardware units to as great a degree as practical. Following this guideline helps develop one's understanding of the relations between various networking components and also encourages one to have better reasons than mere convenience for combining functions into a unit. The analogy with software architecture lies in the way that control functions are represented by processes as discussed below.

The switching control job can be divided into distinct parts: (1) pathfinding, (2) plugboarding, and (3) service management. *Pathfinding* denotes routing, low-level address translation (in the sense of section 3.1.2), and any other operations related to the representation of virtual circuits in the hardware. If we think of the packet switch as a plugboard and virtual circuits as patchcords, then *plugboarding* denotes the relatively simple operation of maintaining the routing table memory in the packet switch. Lastly the term *service management* refers to the higher-level aspects of providing network service. Examples include authentication mechanisms, terminal control, process coordination, resource management, and various kinds of directory and name translation services.

The software for such things is fairly specialized - no two are alike. On the other hand the plugboard and pathfinding functions are closely related and are needed by all the higher level functions. This is the basis of a functional argument for an architecture consisting of a simple low-level process, in our case the CMC, that provides plugboard and pathfinding services for a group of higher-level processes that in turn provide service management functions for the network users.

After the addressing structure has been designed for a network, the services provided by a CMC need not change. Service management by contrast is where one expects to find the unexpected. New kinds of intelligent interfaces, software applications, and combinations of new and old technologies emerge continuously, requiring adaptation on the part of service management software but not typically in the CMC part. This kind of evolution is a phenomenon observed in telephony, but it happens at an even greater rate in computer networking. If there is anything at all to be learned from past experience with ESS software it is that the memory, cpu, and complexity aspects of evolving service offerings present non-trivial problems for the implementors and maintainers. The facts of life seem to be that service management software must evolve and that in doing so it quickly reaches the limits of the small computers favored as control processors. The CMC design advocates placing different service management processes on different, perhaps dedicated, computers so they can evolve without affecting the memory or real-time properties of other processes. Since switching software would probably be partitioned into plugboard/pathfinder and manager functions anyway, physical separation can be achieved by having communications between the CMC and other control processes travel over the network.

As mentioned above the plugboard/pathfinder functions have been implemented as a program called the *common control*, abbreviated CMC. This program implements the low-level *request/response* handshakes for virtual circuit control in support of user programs and service management processes, the latter simply called *managers* or *manager processes*. The CMC also provides an addressing mechanism for manager processes so that programs accessing the network can utilize the higher-level services.

1.4 Configuration and Addressing

The hardware design of Datakit permits circuit board installation and removal while the equipment is operating. This in turn allows maintenance and configuration operations to proceed without turning off the network and also means that the topology of the network can change dynamically. One goal of the software has been to automatically track and accommodate topological change. This led to a loose style of interaction between network control processes and host computers wherein host computers announce their presence to the network control software which is then able to determine their physical addresses. In addition network control processes continually execute a distributed spanning-tree algorithm to determine overall network connectivity, host computers inform the network when they are alive and the network acts accordingly.

If the network topology is to change dynamically, then appropriate translations must be designed into the network addressing mechanisms. The notion of having addressing and routing data in each host, exemplified by the ARPANET, uucp and many Ethernets, is difficult to reconcile with the notion of adaptability. The approach that has been worked out here distinguishes between physical (hardware) addressing and logical addressing. Section 3.1.2 defines a hierarchical addressing structure for the network. The logical addresses so defined are incorporated in the *dialout* message structures exchanged with the CMC. The CMC program is responsible for mapping logical addresses to physical locations in the hardware. The spanning-tree and other techniques mentioned above provide the information needed by the CMC to perform address translations.

1.5 Connection Management

The connection setup procedure was designed for speed and simplicity. It uses the minimum number of interprocess messages - one from caller to CMC, one from CMC to destination, and one from the destination back to the caller - for a total of three. The first and third of these messages are sent and received by a user process - this is the simple request/response model outlined in section 1.2. This strategy has consequences for the internal organization of the CMC program, for routing

and path setup algorithms, and for error control of channel setup messages.

The external request/response interface to the CMC implies that everything the CMC needs to know in order to set up a new virtual circuit is contained in the request message. This means that the CMC can be organized internally as a strict transaction-processing program where each message read by the CMC is completely processed before the next message is read. This tends to make the CMC a simple state machine program that can easily be multiplexed over the many channels and processes that it serves. An example of this is given in section 3.8.2. Note that the CMC can be constructed in this manner with some confidence because the manager process mechanism exists to handle more complex interactions that don't fit the request/response model.

The request messages sent to a CMC originate on channels selected by their senders. A CMC takes these 'requesting' channels as the starting points of new virtual circuits and constructs paths through the network to destination addresses. A Datakit network consisting of many packet switching nodes will probably have a CMC process for each node. This means that a virtual circuit request may need processing by more than one CMC before reaching its destination. This is accomplished by having each CMC set up as much of a new virtual circuit as it can and then pass the request message on to the next CMC which either finishes the job or passes it along to yet another CMC. The details of this procedure are given in section 3.2.4 and need not be repeated here. The design concepts are these: (1) there is no essential difference in handling incoming service requests by a CMC whether the messages come from a user process or another CMC, (2) a user process sees the same request/response handshake regardless of the number of control programs participating in connection management, and (3) the distributed spanning tree algorithm mentioned above provides routing information needed to decide *which* CMC should continue a virtual circuit setup.

The three-message connection procedure is simple because there is no explicit error control. Errors are corrected by starting over again while the connection take-down procedure carefully synchronizes the recovery of channels. If one of the connection setup messages is lost because of a network error, the setup attempt will fail. This kind of error is detected by a timeout in the caller, who drops the failed connection and tries again. The act of dropping a connection means going through the connection takedown procedures (see section 3.4) which do have error correction by retransmission. This is not a disadvantage because the same kind of take-down procedures are needed whether or not the connection setup is simplified. The advantages are that the omission of explicit error control on each channel setup message contributes to lower overall delay in setting up virtual circuits, and the CMC software is simplified as well.

1.6 Network Evolution

One can distinguish at least three phases in the evolution of distributed systems. The initial software provides communication between time-sharing computers. These machines are typically equipped with file storage, terminal ports, and tape facilities and operate quite nicely without a network. The next step produces systems that depend on the network in a number of ways but which can still operate in a standalone manner if necessary. In the third phase computers without terminal ports or mass storage are connected to the net. This kind of architecture approaches as a limit the concept of a processor-per-process operating system.

The present state of the art seems to be closer to phase two than phase three. The so-called third phase network is a goal that seems to be shared among many active researchers in the area. The remainder of this document describes the techniques that have been worked out in anticipation of reaching that goal.

2. DK Device Driver

2.0 General

The Datakit hardware presents 512 logical channels to a cpu. Channel 0 is reserved for hardware status reports and should not be used by a cpu. The switch hardware uses a masking technique to bound the number of available channels on each interface, meaning that a driver should expect to handle some number of channels that is a power of 2.

The network control software arranges for the highest even channel on each host interface module to be a virtual circuit to itself, i.e. looped-back channel. The physical number for this channel will be 2^n-2 where 2^n represents the total number of channels available to the module. Because of the channel-masking property of the switch any data sent on channel 510 will actually go out on the loopback channel and return on it. This facility can be used for diagnostics, or to determine how many channels are available. It is also arranged that the highest channel on each host interface, i.e. channel 2^n-1, is mapped to an empty network address. This channel can be used to cause the initial output interrupt needed by some device drivers.

Although there will be exceptions to the rule, most network interface devices and device drivers will be multiplexed — it is a general assumption that software needs to read and write multiple channels simultaneously. Also, the various protocols defined for Datakit depend on certain scanning, editing, and translation operations. Those operations that are best done in a device driver (or device) are described here as channel *modes*. Examples include whether or not to echo incoming data, or to add additional information to incoming or outgoing packets. The driver on a multiuser system must provide a method for dynamically setting and changing modes on a per-channel basis.

On Unix the directory */dev/dk* contains a special file for each Datakit channel supported by the driver. File names are of the form */dev/dk/dkxx* where *xx* starts at 01. If a user program opens one of these files, the operating system returns a *file descriptor*, usually abbreviated as *fd* in this document, that may be used in subsequent i/o calls. Read or write system calls on file */dev/dk/dk01* affect Datakit channel 1, */dev/dk/dk02* maps to Datakit channel 2, etc.

2.1 Framing and Control

Data transmission through the switch is in *packets*, sometimes called *hardware packets* or *chunks* to avoid confusion with higher-level message abstractions. Each packet consists of a 9-bit *channel number* followed by 16 9-bit data bytes. User data is passed through the network as an 8-bit stream with bit-9 on. Packets with all bit-9's on are called *data packets*. If all nine bits of a byte are zero, it is a *null* or *pad* byte and may be discarded. If bit 9 of a byte is zero and the remaining 8-bit field is non-null, then the data byte is a *control* or *meta* character.

A *control packet* concept, distinguished from that of *data packet*, has been used in several protocol implementations. A control packet is defined as a single hardware packet where the first byte of the packet is a control character. Since data is received from the network one 16-byte packet at a time, there is only a small amount of software overhead involved in checking the first byte to see whether the whole packet should be treated as data or control information.

A more general definition of control packet would permit a control character to appear anywhere within a hardware packet and might also permit the bytes following the control character to span more than one hardware packet. This general definition incurs more software overhead since every byte must be examined and messages may have to be assembled from multiple packets. Byte scanning is easy but time-consuming. Message assembly is time-consuming if data must be copied and is usually difficult to implement correctly.

The simple but restricted control packet definition looks attractive for several reasons, but has the disadvantage of depending on explicit knowledge of the 16-byte hardware packet size. The issue of whether or not to take advantage of hardware packet sizes in software is influenced by the following: (1) new Datakit devices and internet connections may fragment packets and insert null bytes, thus hiding the original packet boundaries and foiling the simple control packet algorithm,

and (2) the 16-byte packet size sets an uncomfortably low limit for some protocols.

It is interesting to note that programmers working with Datakit hardware have all chosen a simple control packet format for short term improvements in efficiency. Experience with the resulting implementations has shown that software overheads are prohibitive for applications other than small-scale terminal processing. In other words the short term efficiency improvements gained by the simple control packet format are negligible compared with the demands of remote file access or bulk file transfer applications. In light of this it seems better to use the more general techniques, even though some performance may be given up, and to rely on hardware front-ends and other improvements for performance when needed.

2.2 Ioctl Calls

Special functions and modes in device drivers are usually accessed on Unix via the *ioctl* system call. Although this method is not the only possible one, the functions that were implemented will be described here in terms of *ioctl* calls. It is assumed that these operations can be mapped into *stty/gtty, fcntl,* or other primitives on systems that don't use *ioctl.*

The following paragraphs describe various *ioctl* calls used with Datakit. The arguments to *ioctl* are a Unix file descriptor for a Datakit channel, ioctl code, and address of a formal parameter. The ioctl codes are defined in */usr/include/sgtty.h* on Unix and have the prefix DIOC. Those calls not requiring a third argument show a – in that position. In all the calls where the third argument is a channel number, usually designated by the letter *x*, the variable is meant to be a 16-bit integer.

2.2.0 Call Mode

The common control program (CMC) for the network sets up the Datakit switch so that messages sent from a host on an idle channel will go to the CMC. All of the idle channels on each host are mapped to the same channel at the CMC. When a message is sent on one of these channels, the CMC must determine which channel the message represents. Obviously a channel number must appear in such messages.

The format of CMC control messages (section 3.1) was designed so that the channel number appears as the first item. A device driver can affix channel numbers to the beginning of output messages – a channel being operated in this manner is said to be in *call mode*.

ioctl(fd,DIOCSCALL,-) turns on call mode for the channel represented by file descriptor *fd*. In this mode every packet sent on the channel is preceded by two extra 9-bit bytes placed there by the driver. In the current definition of call mode these two bytes are simply the channel number, low-order byte first. Note that call mode is turned on automatically when the channel selection mechanism in the driver is used (see section 2.3).

ioctl(fd,DIOCRCALL,-) turns off call mode on the channel represented by *fd*.

It is worth noting that the format of CMC messages does not imply that there be a call mode implemented in the device driver. All that is required is that the channel numbers in messages be accurate. If the channel number in a network control message is correct but the rest of the message is nonsense, the only adverse effects will be on the program sending the message. If the channel number is wrong as well, it may represent an active channel, and some otherwise correctly running program may be affected. This kind of reasoning leads to having call mode or an equivalent in the device driver or operating system where it can be insulated from malfunctioning user programs.

2.2.1 Listener Calls

The connection management protocols in section 3 are defined between three entities: a user process, the network common control, or CMC, process, and an entity called the *listener* at each destination network address. The listener abstraction can be implemented as a single process as was done on Unix, or it could be done in other ways. The driver controls listed in this section were provided for the specific support of the Unix listener process.

ioctl(fd,DIOCLSTN,-) tells the driver that the channel represented by *fd* is to be the *listener* channel for the host (see section 3). DIOCLSTN puts channel *fd* in call mode for sending messages

to the CMC and conditions the driver to send a timeout message to the listener every 15 seconds. The syntax of a timeout message is (T_LOC, D_TIMER)[-] using the notation introduced in section 3.1.1. If *ichan* mode is turned on (see next section), channel numbers are included with received messages. Timer messages and other communications from the device driver are made to appear on channel -1. If the listener process stops executing, the driver clears an internal flag that normally indicates which channel the listener is using. The driver inhibits outgoing channel setups (see 2.3) if a listener is not present.

ioctl(fd,DIOCICHAN,-) causes the driver to insert an 8-bit channel number and 8-bit length before each object delivered to the io stream represented by *fd*. This lets a program distinguish between data on different channels (see DIOCMERGE below), recognize messages generated by the driver, and correctly parse variable-length messages in a byte stream.

ioctl(fd,DIOCPGRP,-) makes the current process the head of a process group and channel *fd* on Datakit the *control typewriter* for that group. The control typewriter mechanism in Unix provides for the sending of asynchronous signals, such as HANGUP, to a group of processes.

ioctl(fd,DIOCRESET,&x) flushes all buffered data on channel *x* and sends a unix signal. The operation varies when the value of *x* is negative, zero, or positive. If *x* is positive, a *hangup* signal is sent to the process group for channel *x*. If *x* is zero any processes (other than the listener) are terminated with extreme prejudice — i.e. the Unix SIGKILL signal is sent to each one. If *x* is negative, data is flushed for channel *-x* and SIGKILL is sent instead of SIGHUP. The process using DIOCRESET must either be the root, or *fd* must be the listener channel. Except for the case where *x* is zero, DIOCRESET sets a driver-internal state (DKLINGR) which prevents channel reuse until DIOCLOSE is issued for the channel.

ioctl(fd,DIOCLOSE,&x) clears internal driver states and modes for channel *x* including DKLINGR.

ioctl(fd,DIOCLOOP,&m) sends the message *m* to a local channel. The channel number is determined by the first 16-bits of *m*. The remainder of *m* is copied to the designated channel. The listener uses this facility to pass error and other messages received from the CMC to local processes.

2.2.2 CMC Calls

The calls grouped in this section provide the driver support necessary to run the CMC program as a Unix process.

ioctl(fd,DIOCMPX,-) turns on user-multiplexing mode: the first 16-bits of each *write* operation to the driver are interpreted as the hardware channel to send data on.

ioctl(fd,DIOCNMPX,-) turns off multiplexing mode.

ioctl(fd,DIOCMD,-) causes output formatting as required for reading and writing a Datakit switch control memory.

ioctl(fd,DIOCTIME,-) turns on 15-second (T_LOC,D_TIMER) time ticks for the CMC on channel -1 as described earlier for the listener. This call dates from the earliest implementations when the CMC program ran on a machine that also had a listener.

2.2.3 Input Multiplexing

A primitive multiplexing ability exists in the Unix driver that lets a process receive data from several channels by reading on a single file descriptor. This was provided mainly for experimentation with various network control processes on Unix.

ioctl(fd,DIOCMERGE,x) instructs the driver to send all incoming data on channel *x* to file descriptor *fd*. DIOCICHAN mode should be turned on in order to decode the incoming stream.

ioctl(fd,DIOCUMERGE,x) turns off the merging operation on channel x.

2.2.4 Other

These primitives permit user program manipulation of control packets and systems buffers. They are described here for completeness only and have in fact been used primarily as diagnostic and development facilities.

ioctl(fd,DIOCBLOCK,-) conditions the driver to use system buffers for i/o operations on the indicated channel.

ioctl(fd,DIOCSMETA,&message) sends the 16-byte *message* as a control packet. The driver flushes all non-control buffers before sending *message*.

ioctl(fd,DIOCXMETA,&byte) sends *byte* as a single control character.

2.3 Channel Zero

Since channel 0 is dedicated to hardware maintenance, the driver uses it for a special purpose: a software open operation on channel zero is interpreted as a request to select and open an available channel. The channel that is actually opened is placed in call mode with echo and other Unix typewriter modes turned off (i.e. RAW mode). Channels that are in use or are in DKLINGR mode are bypassed in the search. This operation can be done very quickly in the driver at high priority thus avoiding the race conditions and overhead inherent in other implementations.

Channel allocation rules, i.e. the definition of what constitutes an available channel, are described in more detail in section 3.2.2. These rules may change from time to time, but we expect to always have the current selection rule carried out as a side-effect of opening channel zero on the driver.

2.4 DR11C Specimen Driver

The following explains the programming steps needed to operate the DR11C-based Datakit interface. The style will be to present and explain fragments of C code that deal with the hardware. The problem of integrating these operations into a driver and particular operating system are left to the reader. The viewpoint of the text is that *input* implies data movement from network to computer.

2.4.1 Hardware Description

A DR11C, or its Q-bus equivalent, provide a register interface to a Computer Port Module, or CPM, that is inserted in the Datakit backplane. The device register layout is:

```
struct device {
        short csr;      /* control/status register */
        short dko;      /* output register */
        short dki;      /* input register */
};
```

The Datakit CPM board may be thought of as an arrangement of three fifo's: input, output, and sequence. Software must load the low-order two bits of the DR11C *csr* register with values that control which fifo is accessed when the software reads or writes *dki* and *dko*.

Packets from the network are copied directly to the hardware input fifo (256 bytes deep) where they propagate to the top and can be read out via the input register *dki*. The output fifo (64 bytes deep) is loaded by writing to the DR11C output register *dko*. The software must delimit packets in the output fifo by giving the hardware a transmit command at least every 16 bytes. Each such command is accompanied by a 4-bit *sequence number*. For each packet transmitted by the hardware from the output fifo to the network, the associated sequence number is written to the sequence fifo where it can be read back by the software.

Bit definitions used in accessing the *csr* register are:

```
#define    DKTENAB    0100       /* enable output interrupts */
#define    DKRENAB    040        /* enable input interrupts */
#define    IENABS     0140       /* both interrupt enable bits */
#define    DKTDONE    0200       /* transmitter done bit */
#define    DKRDONE    0100000    /* receiver done bit */
#define    D_OSEQ     0          /* read  sequence fifo */
#define    D_READ     1          /* read input fifo */
#define    D_WRITE    2          /* write output fifo */
#define    D_XPACK    3          /* transmit packet */
```

The first five definitions are standard ones for a DR11C (see DEC manuals). The DKRDONE bit is set by the hardware whenever there is a data byte to be read from the input fifo. The DKTDONE bit is set by the hardware whenever there is a value to be read from the sequence fifo. The hardware generates an interrupt through the input interrupt vector when both DKRENAB and DKRDONE are set, and vectors through the output vector when DKTENAB and DKTDONE are set. The last four values are *commands* that are loaded in *csr*.

Bit definitions used when accessing the *dki* and *dko* registers are:

```
#define    DKMARK     01000          /* marks beginning of packet */
#define    DKDATA     0400           /* bit 9 of data byte */
#define    DKPERR     0100000        /* input parity error */
```

The first byte of a packet is marked with DKMARK and is interpreted as a *channel* number. Bytes are deemed user data if DKDATA is set. They are deemed control information if non-null and DKDATA is not set. Zero bytes are null pads and may be discarded on input. The DKPERR bit should be *set* for each input byte. A parity error is indicated if DKPERR is *not* set.

2.4.2 Interface Initialization

The following magic will clear all hardware fifo's on the CPM board:

```
csr = D_OSEQ;
dko = 0;
```

It is prudent to test the sequence fifo after this initialization: if the DR11C is not physically connected to the network or if the network is powered-down, the interface will deliver a bogus and infinite stream of data from the sequence and input fifo's. It is important for a Datakit device driver to check for seemingly infinite junk coming from the two fifo's to avoid crashing a host. Code to perform this check for the sequence fifo is shown below. Since it is known that the sequence fifo is 64 bytes deep, it suffices to see if more than 64 bytes can be read from this fifo.

```
csr = D_OSEQ
for(i=64; i; i--)
     if ((csr&DKTDONE)==0)
          break;
if (i==0) {
     printf("bad dk interface");
     return;
}
```

This test works because the sequence fifo is affected only by data traffic going to the network, and software can make certain that the sequence fifo is not being filled at the same time the check loop is emptying it. The input fifo is not similarly independent of network activity. A partial solution to the input fifo problem is mentioned in the next section.

2.4.3 Input Processing

As data enters the CPM board, software can either receive an interrupt (by having set DKRENAB) or loop while testing the DKRDONE bit. The contents of the input fifo will be a stream of 16-byte packets each preceded by a channel number. The channel number bytes are marked with the DKMARK bit. In order to read these bytes from *dki*, software must have first loaded *csr* with the value D_READ. If the software is structured so that read routines call write routines, it is important to make sure that the value D_READ in the *csr* is not changed when reading commences anew. A specimen interrupt handler for input is given below.

```
dkrint()
{
int chan, c, i, j, first;
int save;
char buf[16];

     save = csr;                         /* save csr on entry */
     while (csr&DKRDONE) {
     scan:
          csr = D_READ|IENABS            /* set up for reading */
          do {                           /* find packet header */
               chan = dki;
               if (chan&DKMARK)
                    break;
          } while (csr&DKRDONE);
     begin:
          if ((csr&DKRDONE)==0)
               break;                   /* no more data */
          if ((chan&DKPERR)==0)         /* parity error in header */
               goto scan;
          chan &= 0777;
          if (chan==0)                  /* error: channel zero */
               continue;

          /*
           * Copy out 16-byte packet for channel chan
           */
          for (i=j=0; i<16; i++) {
               if ((csr&DKRDONE)==0)/* error: no data */
                    goto out;
               c = dki;                 /* get a byte */
               if (i==0)                /* remember first byte */
                    first = c;
               if (c&DKMARK) {          /* error: short packet */
                    chan = c;
                    goto begin;
               }
               if ((c&DKPERR)==0)       /* parity error */
                    goto scan;
               if ((c&0777)==0)         /* ignore NULL */
                    continue;
               buf[j++] = c;
          }
          if (j==0)                     /* empty packet */
               continue;
          if (first&DKDATA)
               pass_data(chan, buf, j); else
               pass_cntl(chan, buf, j);
     }
out:
     csr = save                         /* restore csr */
}
```

The error strategy in *dkrint* consists of throwing away bad data. The data/control packet definitions of section 2.1 are applied, meaning that control packets are recognized by inspecting the first byte. Null bytes are discarded. Good packets are accordingly passed to either a data packet handler or control packet handler.

The code displayed above illustrates all of the error checks that should be applied to input bytes except for one. That involves checking for the infinite data problem mentioned in 2.4.2. The code segment given in 2.4.2 protects against continuous output interrupts, but not continuous input interrupts. The input process can be guarded by prohibiting the use of channel 511 and counting the number of packets arriving on that channel. If a sufficient number of packets arrive on channel 511 we decide the interface is out of order.

There are many ways of recoding the dkrint() routine given above. Note that on a non-cached processor it may be advantageous to unroll the copy loop.

2.4.4 Output Processing

In order to do output the software first loads D_WRITE into the *csr* register, then writes the desired output channel number or'd with DKMARK to *dko*, followed by at least 1 but not more than 16 bytes of data. This is followed by a command to transmit the packet. The *dkwrite* routine displayed below writes a block of data specified by address *buf* and byte count *cc* to channel *chan*. It is assumed that interrupt processing would be disabled during the execution of *dkwrite*. Note that since there is a race condition in the hardware between interrupt enable transitions and the asynchronous arrival of data, i.e. setting of DONE bits, the only safe method of programming the interface requires that interrupt enable bits be set and never changed while the driver is active. For this reason the *csr* manipulations in *dkwrite* show IENABS combined with every command. Also note that the variable *seq* is used in the routine but not initialized. This is because software *must* write *dko* once after every D_XPACK command whether or not it intends to make use of the sequence fifo. Data in this extra write accompanies each packet through the output circuitry and is loaded into the sequence fifo when each packet is actually transmitted to the Datakit bus. The actual value loaded into the sequence fifo by this operation is the 4-bit field beginning at bit 10 of *dko*. Hence the code shown below would load the low-order 4 bits of *seq* into the sequence fifo.

```
dkwrite(chan, buf, cc)
char *buf;
{
int i, seq;

      /*
       * Loop until cc bytes are transferred.
       */
      while (cc) {
            /*
             * Set up for output copy.
             * Send channel number and calculate
             * size of next packet.
             */
            csr = D_WRITE|IENABS;
            dko = chan|DKMARK;
            i = min(cc, 16);
            cc -= i;

            /*
             * Move data, preventing sign-extension.
             * Mark as user data.
             */
            while(i--)
                  dko = (*buf++ & 0377) | DKDATA;
            /*
             * Send the packet; load output seq number.
             */
```

```
            csr = D_XPACK|IENABS;
            dko = seq<<10;
        }
    }
```

A software routine like *dkwrite* is not likely to execute the inner copy loop faster than once per microsecond, i.e. no faster than Datakit can accept bytes. This means that such routines need not worry about output fifo overruns.

On the other hand those software or firmware implementations that are capable of exceeding the bandwidth of the network need to know when packets are moved from the output fifo to the network if fifo overruns are to be avoided. The only way to accomplish this with the DR11C/CPM interface depends on using the sequence fifo: the variable *seq* in `dkwrite()` can be loaded with sequence numbers which when returned by the hardware indicate packet transmission. The hardware algorithm propagates the *seq* values through the output circuitry along with the output packets. When a packet is removed from the output fifo and copied to the Datakit bus, the associated *seq* variable is loaded into the sequence fifo. Output interrupts are caused by the arrival of a sequence number at the top of the sequence fifo. These numbers may be read out of the sequence fifo in an interrupt routine as shown below.

```
dkxint()
{
int save, seq;

    /*
     * Preserve contents of csr, load with OSEQ command.
     */
    save = csr;
    csr = D_OSEQ|IENABS;
    /*
     * Unload sequence numbers.
     */
    while(csr&DKTDONE) {
        seq = (dki>>10)&017;
    }
    /*
     * restore csr.
     */
    csr = save;
}
```

Note that the while statement in the above constitutes a potential endless loop. There should be a limiting mechanism for this loop in a production driver along the lines of those discussed in 2.4.2 and 2.4.3.

2.4.5 Observations

The 16-bit parallel interface to host computers was selected to simplify the construction of experimental hardware. This was a wise choice - Datakit was put on the air using off-the-shelf host interfaces. Other consequences of the CPM board and DR11C combination are enumerated below.

A computer would have to deliver a 1-megabyte/second stream in order to overrun the output hardware fifo. It is believed that the software-controlled algorithm given above cannot induce this phenomenon when run on a minicomputer. Oscilloscope timings have shown that a slightly optimized version of the copy loop given above will move a byte approximately every 10 microseconds when executed on a cache-equipped PDP 11/45. The setup code that must be executed between 16 byte bursts introduces additional overhead.

The major performance bottleneck on the current DR11C interface is caused by the inability to move more than 16 bytes through the interface before having to observe a begin-packet or end-

packet protocol with the CPM board and other software components. In other words the software spends most of its time setting up 16-byte copy loops. The second performance problem has to do with the amount of scanning required to check the incoming byte stream plus the overheads caused by having to demultiplex the interleaved input packet stream. The third performance-affecting property of the CPM board is the size (256 bytes) of the input fifo: long messages are not practical, and slow or heavily loaded hosts can experience fifo overruns.

The DR11C forces a half-duplex discipline in accessing the fifo's on the CPM board. This causes minor irritations in a software driver and induces a small processing overhead.

The lack of electrical isolation between DR11C and CPM board plus the bad effects of Datakit power-off conditions on the DR11C are problems we can live with in a research lab but cannot be ignored in new designs. The problems are caused by the DEC DR11C and seem to be avoidable only by modifying or replacing it.

Experience with the DR11C/CPM combination has been that the architecture provides a convenient abstraction of virtual circuits and is exceptionally easy to program, especially when compared with other network and communications hardware. The various components of i/o performance have been easy to observe because the interface is simple, and the knowledge gained contributes to new designs that will improve performance without perturbing hardware costs very much.

3. Network Control

3.0 Overview

A full-duplex end-to-end data path between a pair of objects in Datakit is referred to as a *virtual* circuit. The following processes participate in managing virtual circuits:

1) CMC - the common control program; each CMC in the network is identified by an *exchange number;*
2) L - the listener program, or datakit server, which runs on each computer; each listener is identified by a number referred to as a *logical address, logical host number,* or sometimes *local address,*
3) U - a user program initiating a request for a virtual circuit on some computer in the network;
4) DK - the datakit device driver;
5) MGR - a manager process responsible for some network resource or service.

Protocols are defined for the pairs of processes U/CMC, CMC/L, CMC/MGR, DK/L and CMC/CMC. The protocols for virtual circuit maintenance are command/response exchanges. That is, one process sends a command to another process and expects a response within a certain amount of time - typically 15 seconds. Commands are differentiated as to whether an incorrect or missing response is fatal or whether a command should be retried until it succeeds. In general commands for setting up connections may not be retried without first closing the channel, whereas commands for closing a channel are to be repeated until a proper response is received.

3.1 Formats and Conventions

3.1.0 Data Representation

Unless otherwise noted bit zero and byte zero are the least significant elements in whatever context they appear. Serial transmission of message structures is in increasing order of significance starting with byte zero. This means that messages are copied to the network in the same way the human reader scans this text: left-to-right and top-to-bottom. A pair of subroutines that convert between the canonical, or network transmission, representation and the local data representation of a machine are described in section 8. An explicit example of a C message structure and its canonical byte representation is given in section 3.1.1 below.

3.1.1 Message Formats

All message structures and related definitions are in */usr/include/dk.h*, and are accessed by the cmc program and routines that talk to the cmc with the line *#include <dk.h>*. All messages are based on the *dialout* structure:

```
struct dialout {
        char type;
        char srv;
        short param0;
        short param1;
        short param2;
        short param3;
        short param4;
        short param5;
};
```

The representation of C structures in memory tends to differ between computers and C compilers. Section 3.1.0 defines how messages are to be formatted for transmission. The diagram below shows the transmission byte ordering for the *dialout* structure.

type	srv	p0.l	p0.h	p1.l	p1.h	p2.l	p2.h	p3.l	p3.h	p4.l	p4.h	p5.l	p5.h

Transmission is from left to right; *p0.l* and *p0.h* denote the low and high bytes of param0. When messages like the one above are sent in call mode (section 2.2.0) a channel number is required at the beginning. This is normally put on by the device driver as depicted below where *ch.l* and *ch.h* are the low and high-order bytes of the channel number that the message is sent on.

ch.l	ch.h	type	srv	p0.l	p0.h	p1.l	p1.h	p2.l	p2.h	p3.l	p3.h	p4.l	p4.h	p5.l	p5.h

The *type* and *srv* in a message structure determine the interpretation of the six 16-bit parameters. The notation (type,srv)[p0,p1,p2,p3,p4,p5] will be used to describe messages. For example (T_SRV,D_SH)[*arex,host*,-,-,-,-] denotes a message of *type* T_SRV and *srv* D_SH with *param0* and *param1* indicating respectively an area code and host number. The dashes signify that the other parameters are not defined. Messages may also be denoted by the abbreviated form (*type,srv*) when the parameters can be omitted from the discussion. Standard set notation, e.g. {a,b,c}, will denote the set of entities a, b, and c.

The listener and common control program use DIOCICHAN mode (see 2.2.1). Thus messages read by the listener have the following form:

```
struct listenin {
        char l_chan;
        char l_size;
        struct dialout l_dial;
};
```

In this structure *l_chan* and *l_size* are the extra bytes put on the message by the DIOCICHAN mode in the driver. The *dialout* structure will be received in canonical order as described above, meaning that some conversion is needed when the local representation of message structures is different from the canonical one.

All messages sent to the common control are formatted in call mode, so the CMC input records are as follows:

```
struct dialin {
        char i_chan;      /* local channel number */
        char i_size;      /* message size */
        short i_rchan;    /* remote channel number */
        struct dialout dial;
};
```

where *i_chan* and *i_size* are supplied by the CMC's device driver and the rest comes from the machine sending to the CMC. Note that *i_rchan* is the manifestation of call mode on the machine sending to the CMC. The notation <chan,size>(type,srv)[p0,p1,p2,p3,p4,p5] will be used in a few cases where it is necessary to discuss *dialin* messages.

3.1.2 Network Addressing

Addressable objects in the network may be roughly categorized as either *physical* or *logical* objects. The physical addresses of interest are either module numbers, i.e. backplane slot positions where hardware modules are inserted, or node numbers, i.e. the identification of particular nodes within a cluster of switches controlled by a single CMC. Logical addresses refer to computers, CMC's, and other resources. Network addressing is based on a three-level hierarchy of logical addressing. The terminology is as follows:

a *local address* is a 16-bit number identifying a host computer or other resource;

an *exchange code* is an 8-bit number identifying a CMC process;

an *area code* is an 8-bit quantity identifying a geographic location, e.g. MH, IH, HO;

a *network address* consists of a host number, plus an exchange and area code.

An exchange and area code are usually combined as a 16-bit quantity, exchange as the low-order byte, the whole referred to as an *arex*. When the value zero appears as any of the three components of a network address it is always interpreted as the *local* or *current* exchange, area, or host.

The common control, or CMC, program maps network addresses to physical addresses. The translation from other representations of system services or resources, such as system 'names', to network addresses is to be carried out by other network processes.

3.1.3 Standard Reply Message

Almost all interactions with network control software are of the command/response variety: a program sends a command and waits for a response. All responses are of the form (T_REPLY,*x*)[ch,err,-,-,-,-] where *x* is one of D_OPEN, D_ACK, D_WAIT, or D_FAIL. The OPEN response is generally made for commands that create new virtual circuits, ACK is a general acknowledgement, WAIT means wait, and FAIL means something went wrong. The value of *param0* in T_REPLY messages sent by the CMC is always the channel number for which the message is intended. The *param1* field accompanying WAIT specifies a number of seconds by which the requesting process should lengthen any timeouts. For FAIL messages *param1* is an error code and may be used to index the array *dkmsgs* defined in */usr/include/dkerr.h*.

3.1.4 Message Framing

All of the network control messages are designed to fit within a 16-byte Datakit packet. This does not include any additional information that might be added by a receiving device driver. Thus the control message framing is built on the 16-byte underlying packet structure provided by the network hardware. All of the CMC and related network control software depends on having a one-to-one relationship between hardware packets and software control messages.

The hardware-oriented message framing simplified the first generation design and implementations. The alternative would have been to implement a full-blown demultiplexing and error-detecting protocol just to set up virtual circuits. This seemed like an inadvisable overhead when the software was first put together even though the price paid for simple software was fixed size limits on the messages.

The principal drawback of the binary formats described above is that it not possible to include an arbitrary length string of bytes, e.g. a computer 'name', or file name, or executable string, as part of a network control message. Such strings must first be converted to a 32-bit network address as mentioned in section 3.1.2. Improved versions of Datakit software currently being implemented use an error-correcting flow-control protocol similar to the trailer protocol described in section 5 of this memo. When this software is in place on network control channels, the message structures can be extended.

Given a variable-length control message containing one or more string variables, the translation of a string to a 32-bit network address can be modeled as part of the channel setup protocol. If a string were part of the channel setup message, the CMC could parse it to obtain a network address. In a more ambitious setting one could imagine the network control protocols being entirely ascii - no binary message formats at all. In this setting the contents of a service request message would change as the message passed through network control processes on its path from source to destination. Higher-level network services are more readily captured by this gradual string translation model than by protocols with multiple messages. However demonstrating this remains the province of future software implementations.

3.2 Standard Connection Setup Procedures

3.2.0 General

A connection, or virtual circuit, is a full-duplex path through the network that connects a pair of cooperating processes. Before a new connection can be made there must exist a way for the requesting process U to contact the common control CMC that makes connections in the Datakit hardware. This is provided for by having a default destination for each channel: each odd-numbered channel on a module is routed to the CMC (by the CMC at boot time), and each even channel is mapped to a non-existent address. The idea is that any software process can send a message to the CMC by transmitting on an odd channel that is in the default state. Similarly the CMC can allocate from the even number channels to create a virtual circuit between a requesting channel and a destination in the network. The odd-even separation eliminates collisions between outgoing messages to the CMC (odd channels) and new connection setups (even channels).

A connection is created by the following sequence of events: (1) a process U obtains a local channel; (2) U sends a *service request* message (T_SRV) to the CMC; (3) the CMC forwards the service request to the appropriate network host if possible and sets up a virtual circuit between U's channel and a channel selected by the CMC on the destination machine; (4) the listener process L on the destination machine processes incoming service requests, sending acknowledgement messages on the new channel to U (or to the CMC if requested). The three interprocess messages that set up a virtual circuit are depicted below in Figure 3.2.1. These and other related messages are examined in detail in the next few sections.

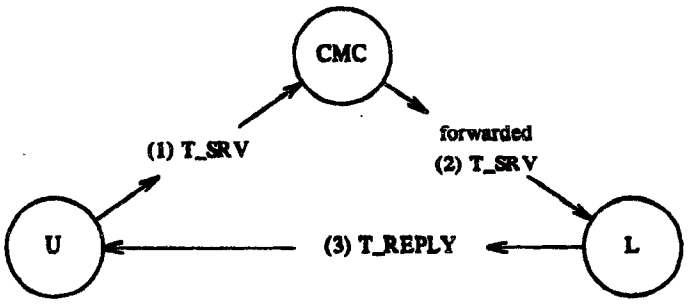

Figure 3.2.1

There are three failure events for the message sequence depicted in Figure 3.2.1. These are: loss of any of the three messages, rejection of the T_SRV message by either of the CMC or Listener, and elapsed time alarm at the U process while waiting for T_REPLY. The rejection event is represented by a T_REPLY message carrying the D_FAIL parameter (see section 3.1.3). Message loss is detected by timer in U. In all failure modes the channel used by U must go through the channel takedown procedure (section 3.3) before it can be used again. The channel takedown insures that each channel is put into an 'intermediate' state while CMC and host software exchange handshakes prior to declaring a channel 'idle' and available for reuse.

It should be pointed out that the listener process, L, user process, U, and other processes discussed below are convenient *abstractions* for relationships and operations that take place during the course of setting up and using virtual circuits. The connection protocols are described in terms of these abstractions, but it is important to note that the protocols are independent of the actual process structure on a host computer. This means, for example, that the messages attributed to the Listener during channel takedown could be implemented in a device driver or in a front-end

computer.

3.2.1 L/CMC

Except for maintenance procedures, the CMC will not set up virtual circuits between a pair of network addresses unless there is a listener or manager process running somewhere in the network on behalf of each host. The reason for this is that at least one process per host is needed that will take responsibility for synchronizing channel states with the CMC. A special case is defined for single-user single-process operating systems in order to reduce software requirements.

When a listener process starts up, it obtains an outgoing (odd-numbered) channel from the operating system/device driver. This channel, called the listener's *cmc channel*, is used for L/CMC and CMC/L communications for the lifetime of the listener. Note that the listener's cmc channel is always in call mode.

Each active L in the system sends a keep-alive message to the CMC every 15 seconds. This is depicted in Figure 3.2.2. The listener's channel to the CMC must be in call mode (2.2.0). If two messages are missed the CMC assumes that the host, or at least its network software, is dead. The form of the message is (T_LSTNR,0)[*lad*,-,*mode*,-,-,-], where *lad* is the listener's local address, and *mode* is a bit field that controls how the T_LSTNR message is interpreted at the CMC. If the P_RESET bit is set in *mode*, the CMC will clear any virtual circuits currently active on the module sending the T_LSTNR message. If the P_ACK bit is set in *mode*, the CMC will return a T_REPLY message. This provides the listener with a means for testing whether the network is alive.

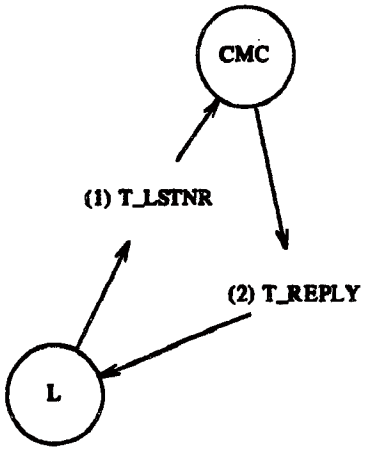

Figure 3.2.2

Constraints on T_LSTNR messages checked by the CMC include: *lad* in T_LSTNR message is in use by another host, or there exists a listener process for the host, or *lad* supplied by L disagrees with any topological knowledge that may be built into the CMC. The CMC will attempt to send (T_REPLY,D_FAIL) if any of these errors is noticed, but this may not always be possible.

The importance of the listener and the L-related protocols diminishes somewhat for single-user/single-process computer systems. Such systems indicate their nature to the CMC by sending (T_LSTNR,1)[*lad*,-, *mode*,-,-,-] when coming on the network instead of (T_LSTNR,0). The CMC will reply to this message exactly as described above. However, such systems are exempted from sending the T_LSTNR message at 15-second intervals. The channel takedown protocol is also optional: a single-user host may ignore CMC-initiated channel takedowns (the CMC will give up after a while), and it may neglect to close any channels it uses. Receipt of a subsequent T_LSTNR message from the host with P_RESET set in the *mode* field will clear any channel setups left hanging from a previous network access. For a minimum software implementation, the only essential network control message other than T_LSTNR is the T_SRV message for requesting network service (see next).

3.2.2 U/CMC

The protocol is command/response: U sends (T_SRV,*srv*) to the CMC and expects a standard success/failure message (see 3.1.3) in return. Figure 3.2.1 shows the T_SRV message going to the CMC and the reply T_REPLY coming from the destination. This is the normal case, although in some failure situations the reply may be sent to U by the CMC. The main example of the latter type of failure occurs when the service requested by U is not available.

The precise message format is (T_SRV,*srv*)[*arex*, *lad*, *mode*, -,-,-] where *srv* is taken from the list of network services (see section 4), *arex* and *lad* specify a network address (see 3.1.2), and *mode* is treated as a bit field for selecting options. At present no bits are defined in *mode* for user processes. When a (T_REPLY, D_OPEN) reply is received (refer to 3.1.3) in response to a T_SRV service request, *param2* of that message will contain the destination machine's channel number for the new virtual circuit.

As mentioned above the channel allocation scheme initially connects all even-numbered channels to a non-existent address; odd-numbered channels are connected to the CMC. To obtain network service, i.e. set up a virtual circuit, a user process first obtains a free odd channel in call mode on the local machine, sends a T_SRV message on that channel and waits for a reply. On Unix these operations are accomplished by the library routine *dkdial* (see section 8).

3.2.3 CMC/L

The CMC forwards a T_SRV request to the listener's cmc channel on the destination machine. This happens after the CMC has set up a new virtual circuit between the calling channel and a new (even-numbered) channel selected by the CMC at the destination address. The forwarded message is (2) in Figure 3.2.1. The content of the message is (T_SRV,*srv*)[*arex*, *lad*, *n*, *t*, *carex*, *clad*] where T_SRV and *srv* are from the user, *arex* and *lad* identify the called machine or terminal, *n* is the (even) channel number on which *srv* is requested, *t* selects where L is to reply (see below), and *carex* and *clad* are the network address of the caller. If *t* is zero, the reply is sent on channel *n* as shown in Figure 3.2.1, otherwise the reply is sent to the CMC on the listener's CMC channel. This ability to redirect the T_REPLY messages was originally provided so that the control program for terminal multiplexors could receive T_REPLY messages rather than have them sent to the terminals.

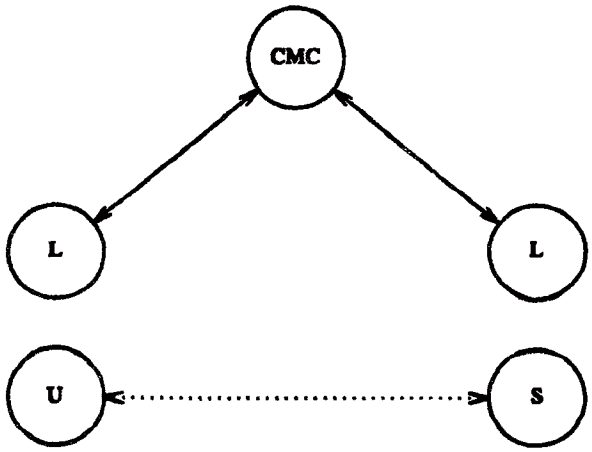

Figure 3.2.3

After the listener acknowledges a connection setup it is responsible for providing the requested service. This usually consists of starting a server process S and giving it access to the new connection. While the connection is in use the state of affairs is as depicted in Figure 3.2.3. It is assumed that U and S are on different computers. The fact that there is a listener process in contact with the CMC on behalf of each machine is shown explicitly.

The listener may reject a service request by sending T_REPLY with a D_FAIL parameter instead of D_OPEN. The listener may also do nothing, ignoring a service request. In this case the requesting process will time out, closing the channel.

3.2.4 CMC/CMC

A CMC can control more than one node by accessing the control memory of remote switches via trunk channels. It is more common for there to be one CMC process per node. In this case CMC's must cooperate in setting up connections between nodes. This section explains how routing information is exchanged between CMC's and how connections are set up across trunks.

The operation of Datakit trunks is such that data received at one end of a trunk on some channel *i* is carried through the trunk and retransmitted at the module on the 'other' end also on channel *i*. A trunk connecting two nodes is depicted in Figure 3.2.4. Packets can flow from any module in node 1 to switch SW1, to trunk T1, through the trunk to module T2, to switch SW2 and thence any module in node 2. The node 2 to node 1 flow is similar.

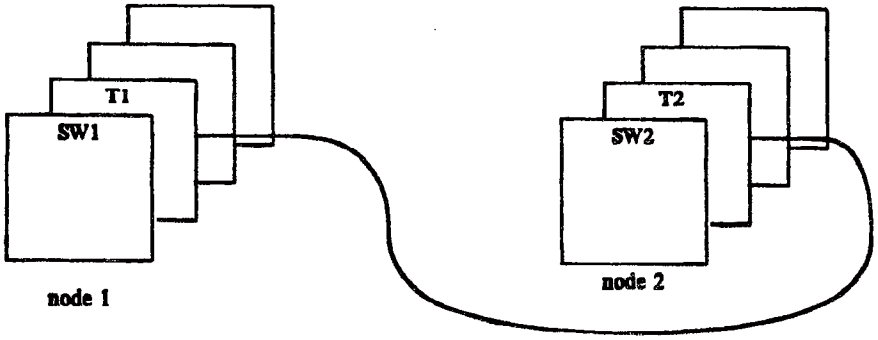

Figure 3.2.4

A pair of CMC's connected by a trunk exchange control messages on trunk channel 5. Channels 1 through 4 are reserved for diagnostics and testing. In particular channel 1 is connected to channel 2 on each end of a trunk. These are half-connections; i.e. channel 1 sends to channel 2, but the converse is false. This provides a loop-around facility for testing trunks during normal network operating conditions. Channel 4 is reserved for routing channel-zero maintenance data to a network monitoring station (see section 3.4.6).

Each CMC sends the other (T_CMC,0)[*arex*,0, *mode*,*links*,-,-] every 15 seconds where *arex* is the area code and exchange for that CMC, *mode* is a bit field explained below, *links* is part of the routing algorithm described below, and the last two fields are reserved for traffic information.

Bits defined for the *mode* field are P_RESET, P_ACK, and P_NMS. The P_RESET and P_ACK bits are similar in operation to those described for listeners in 3.2.1. The response to (T_CMC, 0) with P_ACK set is (T_CMC,D_ACK)[arex,n,-,-,-,-] where *arex* identifies the CMC sending the message and *n* is the maximum number of trunk channels that the responding CMC is prepared for. The response to the P_RESET bit is to take down all active channels on the trunk. The P_NMS bit notifies the CMC receiving it that there is an active status-monitoring unit on the sender's side of the trunk. The receiving CMC may choose to route all its hardware status channels (channel zero on each module) to channel four on the trunk, knowing that the opposite CMC will direct the traffic to a monitoring system (see also section 3.4.6).

When a CMC receives (T_CMC,0) on a trunk, it increments the value of *links* and broadcasts the revised message on all its trunks except those on which the T_CMC message for *arex* has already been received or if *arex* is that of the CMC. This procedure has the effect of distributing the arex code of each CMC to every other CMC in the network every 15 seconds. Each CMC should receive a message for every distinct path between it and another CMC. The length of each distinct path is found in *links*, and the current traffic load on that path will in future be obtainable from the remaining fields in the message.

Connection setups across trunks proceed as in Figure 3.2.1 except that the single CMC is replaced by more than one as illustrated in Figure 3.2.5.

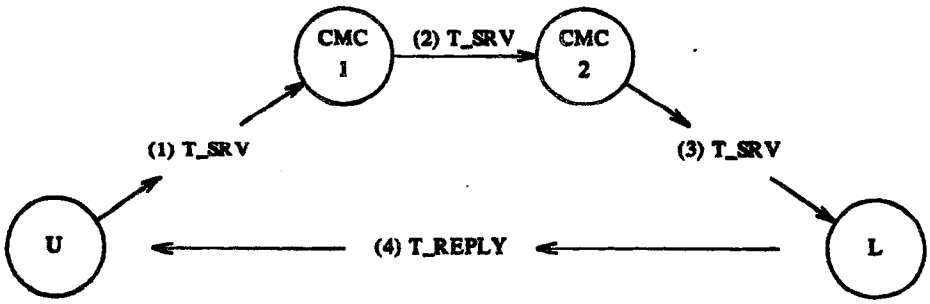

Figure 3.2.5

The protocol used at the U and L ends remains exactly the same: the connection setup protocol requires no topological knowledge, only logical addressing.

The connection setup depicted in Figure 3.2.5 proceeds in stages from left to right. CMC 1 first determines that the connection must terminate in the part of the network controlled by CMC 2. CMC 1 sets up a virtual circuit for U as far as the trunk, informs CMC 2 on channel 5 by sending a T_SRV message, and CMC 2 completes the job by connecting the trunk channel selected by CMC 1 to the destination L. The switch having the numerically greater exchange code allocates the odd-numbered trunk channels; the other switch controls the even-numbered ones. This partitioning of channels permits collision-free forwarding of connection setups from one exchange to another with a minimum (1) number of messages between CMC's.

The connection setup message between CMC's is <*cmcno*, *n*>(T_SRV,*srv*)[*arex*, *host*, *n*, *t*, *carex*, *clad*] where *n* is the new trunk channel, *t* controls the listener's reply, and *carex* and *clad* are the arex and host network address of the connection request originated. These last two fields are loaded by the first CMC to process a service request message and are preserved across trunks and other CMC's.

3.2.5 Summary

A listener process sends (T_LSTNR,0) messages at 15 second intervals to a CMC to maintain its presence on the network. A user process sends (T_SRV,*srv*)[*arex*, *host*, *mode*, -,-,-] to the CMC on an available channel in call mode and waits for a standard reply. A listener receives forwarded T_SRV messages from the network and is responsible for sending the standard reply.

3.2.6 Commentary

The even-odd channel allocation discipline is one of several that solve the contention problem between incoming and outgoing virtual circuit setups. The other most likely candidates are either a top-bottom scheme where hosts allocate from the bottom and the CMC allocates from the top, or schemes where the CMC assigns all channels. The even-odd or top-bottom schemes have the property that a CMC or network host can select channels without having to communicate with another process. This has several advantages in simplifying the model of the state of a channel and in preserving the simple transaction-oriented style of the CMC. Of the two schemes the top-bottom one makes better use of available channel space under asymmetric loaded conditions than the even-odd one.

The first software implementations were simplified by the adoption of the even-odd scheme, and the partitioning of channels catered to the healthy paranoia of the implementer. These considerations have lessened somewhat and might encourage a change to top-bottom strategy.

The network control messages described in this section are not error-controlled. Although errors during channel setup are infrequent they do occur, and it has been interesting to observe what kinds of things go wrong. In general errors in sending T_SRV messages from network host to the CMC are practically nonexistent, and this is also true for messages from the CMC to a listener.

Problems that have been observed seem to involve noise or bad data mixed in with the first messages sent over a new virtual circuit. It is felt that new virtual circuits should be flushed on an end-to-end basis before being used. One way of accomplishing this involves sending T_REPLY as a control message and having software awaiting the T_REPLY message ignore all incoming data except for control messages. A version of the software was tested in which all messages involving the CMC or a listener were sent as control messages and received through a filter. However the actual channel setup error probabilities are low enough that this version was never installed on a permanent basis.

In order to recover from temporary CMC outage or transmission errors in the U/CMC or CMC/L path it suffices for the U process to be able to repeat the transmission of its T_SRV request or to install a low-level error-control algorithm on the channels used for these communications.

Although experience with local Datakit networks suggests that error rates are low, there are reasons to install a more elaborate protocol on channels to the CMC. One reason is that trunk channels are more error prone by nature than virtual circuits within a single Datakit node. The author's CMC process often controls several Datakit nodes through trunks. In this case and in the CMC/CMC case lack of error correction on virtual circuits to a CMC can be a liability since the network control messages pass through a trunk. Another reason is that a low-level protocol would provide flow control as well as error control. Since no flow controls are specified for messages going to a CMC, one can imagine a CMC being 'flooded' with service requests and other messages. A low-level protocol on the so-called *cmc channels* (section 3.2.1), can be used to regulate the aggregate request rate to a CMC by regulating the request rate from each network address. Lastly, a low-level error and flow control protocol is a prerequisite for implementing the control message extensions described in section 3.1.4.

3.3 Connection Takedown Procedures

3.3.0 General

Channel takedown protocols define message handshakes that synchronize changes in the state of a virtual circuit. The sequence of events begins with an active channel. The active state was illustrated earlier in Figure 3.2.3 where two processes, U and S, are depicted at different network addresses each with a listener process. When a process terminates or otherwise relinquishes a network connection, we assume that the local listener process/abstraction is made aware of the change in state. We imagine that the S process shown before in Figure 3.2.3 terminates. The immediate result is diagrammed below in Figure 3.3.1 where the S process has been replaced by a "release channel" symbol 'notifying' the listener. It should be obvious that connection takedown procedures must handle the various cases of one side or the other side of a virtual circuit releasing the circuit first, as well as the case where both sides release at the same time.

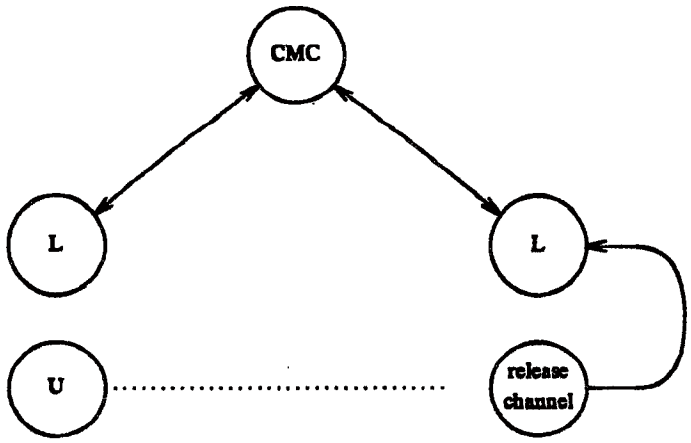

Figure 3.3.1

It is inappropriate for the network protocols to define how the "release channel" information is communicated to the listener, or how anything else is to be done save the details of actual network message exchanges. In the following sections it should be understood that the network protocols are between CMC and L, or CMC and CMC. Message exchanges between the device driver DK and listener document what happens in one existing implementation where the idea was to put as much of the protocol into the listener for purposes of experimentation and debugging. Other implementations carry out the handshakes, attributed below to the listener, in the device driver.

3.3.1 DK/L

The device driver manufactures (T_LOC,D_CLOSE)[*n*,-,-,-,-,-] and sends it to the listener when the last close is done on channel *n*. The listener must have ICHAN mode enabled: driver close messages are labeled as having arrived on channel -1. Although a channel in this state is free, the driver keeps it marked busy until the listener executes a primitive (DIOCLOSE) that 'frees' the channel for other processes. This is done to prevent channel reuse before synchronization with the CMC.

3.3.2 L/CMC

A listener sends (T_CHG,D_CLOSE)[-,*n*,-,-,-,-] to the CMC to indicate that channel *n* has been shut down. The D_CLOSE message is to be sent repeatedly until the CMC responds with (T_CHG,D_ISCLOSED)[-,*n*,-,-,-,-], meaning that L and CMC agree on the state of channel *n*. This handshake between the CMC and L is illustrated by Figure 3.3.2. Note that the CMC breaks the

virtual circuit as soon as a D_CLOSE message is received from either side. When the CLOSE/ISCLOSED handshake has been completed for one side of a virtual circuit, that channel may be reused.

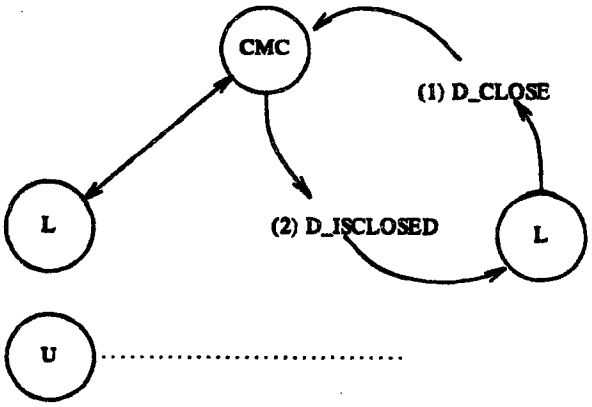

Figure 3.3.2

3.3.3 CMC/L

The procedure described under L/CMC is symmetric in that the CMC initiates a channel shutdown by sending D_CLOSE messages to a listener (same format as above). Upon receipt of such a message a listener takes local action to hangup the channel, eventually leading to a D_CLOSE message from the driver at which point D_ISCLOSED can be sent to the CMC. The CMC will repeatedly send D_CLOSE for a particular channel until the D_ISCLOSED response is received from the appropriate listener. This sequence of events is depicted in Figure 3.3.3.

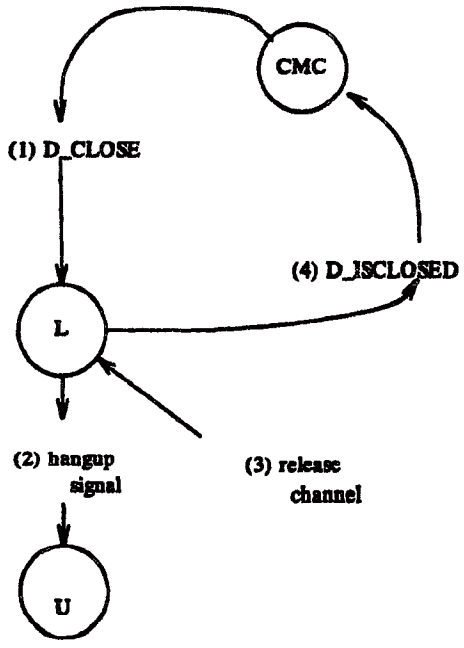

Figure 3.3.3

3.3.4 CMC/CMC

CMC's exchange D_CLOSE and D_ISCLOSED messages on channel 5 as described for L/CMC above. A CMC upon receiving D_CLOSE from an adjacent CMC is responsible for tearing down its part of the virtual circuit. This means either sending D_CLOSE to a listener or to another CMC.

3.3.5 Summary

The L process at each end of a virtual circuit goes through a command/response handshake with its CMC. The initiator sends D_CLOSE until it is acknowledged by D_ISCLOSED. Channel take-down can be started by an L, a CMC, or both.

3.4 Maintenance

3.4.0 Auxiliary CMC Channel

Maintenance and other procedures require allocation of one of the channels that the CMC reserves for the purpose. These are called *auxiliary* channels. A free channel in call mode can be mapped to a CMC auxiliary channel by sending (T_SRV, D_AUX) and waiting for a standard response. The channel, which must remain in call mode, can be used to communicate with the CMC. Maintenance primitives that actively affect the switch can be dangerous in the sense that accidental use could disrupt user communications. For this reason the CMC could be modified to accept D_AUX and other commands only from known network addresses. For non-maintenance processors the CMC could also demand a password after an auxiliary channel has been set up. This would be indicated by responding with D_PLOG instead of D_OPEN. The response to D_PLOG would be a null-terminated ASCII string.

3.4.1 Nailed Channels

Arbitrary connections between modules can be set up by sending (T_MAINT, M_NAIL)[*swtag*, *m1*, *c1*, *m2*, *c2*] on an auxiliary channel to the CMC. The CMC attempts to set up a full-duplex path between module *m1* channel *c1* and module *m2* channel *c2* on the local switch. The *tag* bits in the channel routing table are loaded from (*swtag*>>8). Recall that the tag bits are used to mark command channels (see switch documentation). If the CMC is controlling multiple switches, (*swtag&0377)* selects a switch. After the channel is set, the CMC sends (T_REPLY,D_OPEN) or (T_REPLY,D_FAIL).

A nailed channel is removed by sending (T_MAINT,M_PRY)[*sw*, *m1*, *c1*, -,-,-] on an auxiliary channel to the CMC. This will be acknowledged by (T_REPLY,D_ISCLOSED).

3.4.2 Module Reset

Sending (T_MAINT, M_MBOOT)[*sw*, *mod*, -,-,-,-] on an auxiliary channel will knock down all channels on module *mod*, switch *sw*, within the local exchange.

3.4.3 Switch Reset

Sending (T_MAINT, M_SBOOT)[*sw*,-,-,-,-,-] loads the control memory of switch *sw* in the local exchange with the default settings. This command is used on those occasions when a CMC is controlling several switches and it is desired to bring one of them on line without disrupting the rest of the network.

3.4.4 Configuration

The CMC programs that have been written by various authors assign channel memory space in the Datakit switch to the different hardware modules when the programs begin execution. If a module with a small number of channels is replaced by one requiring more channels, these control programs must be restarted, interrupting network operations, to affect the assignment change. The issue could be eliminated by simply assigning a large amount of channel space to each module. Unfortunately the control memories in the original Datakit switches are not large enough for a large default channel assignment.

A procedure for notifying a CMC of changes in network topology not detectable by the CMC/CMC procedure (see 3.2.4) should be implemented. The goal is to avoid interrupting operations on a Datakit node to effect ordinary configuration changes. When CMC's are able to reconfigure switch control memories, the following proposed message interface could be implemented.

The message interface consists of sending the message (T_MAINT,M_CONFIG)[*sw*, *mod*, *mtype*, *nchan*, -,-] on an auxiliary channel and waiting for the standard response. In this message *sw* is the hardware switch number, *mod* is a physical slot number, *mtype* is the board type inserted in the slot, and *nchan* is the number of channels to be reserved for the board.

3.4.5 Tracing

It is possible to obtain an auxiliary channel to the CMC and receive copies of internal tables and other items. The interface is pretty crude and will not be documented here. However the idea of accessing remote data structures from a program is worth mentioning. Since a process other than the CMC can obtain a so-called *command channel* to a Datakit switch memory, it is possible to verify the correct operation of a CMC by comparing the state of the hardware with the state indicated by the internal tables of the control program. The technique was used by the author as a debugging aid.

3.4.6 NMS

A network monitoring system processor announces itself to the CMC by sending (T_NMS,M_LIVE)[*ch*, -,-,-,-,-] to the CMC on a free channel in call mode. The CMC responds by routing channel zero from every module in the local exchange to channel *ch* on the NMS. It is expected that the NMS software will eventually use the message interface of 3.4.4 to notify the CMC of configuration changes.

A CMC with an active NMS notifies adjacent CMC's of this with periodic messages. It also routes channel four of each trunk it controls to the NMS. This is the channel reserved for remote status traffic (section 3.2.4). This technique has the effect of routing status information to an NMS from every nearby Datakit node that does not have a private NMS. It also has the property that one can move an NMS from place to place in a Datakit network and the channel zero status messages will follow.

3.5 Manager Processes

3.5.0 General

The connection management protocol described in sections 3.2 and 3.3 is called 'low-level' because the process initiating a connection must supply a destination address. In this protocol the CMC process acts on commands received from user processes. CMC operations consist mainly of processing network addresses contained in commands.

There are many applications in networking – distributed file access, address and name translation, authentication, network resource allocation – that require addressing and control mechanisms different from the lower-level one understood by the CMC. As an example of higher-level addressing consider the command "connect to *any* line printer", which implies finding a line printer in the network, as opposed to the specific request to "connect to the line printer server at network address X". For another example consider the problem of establishing file server connections in a network. In many distributed file system designs there are multiple servers at different network addresses and the binding of files to file servers is meant to be transparent. This means that neither the CMC, as it has been defined here, nor a user program can calculate the network address for a file server given a file name. Both of these examples illustrate an important characteristic of 'higher-level' address mechanisms: the address given in the command is implicit and some searching or decision-making computation is needed to make the address explicit, i.e. to *bind* the command to a network address. Schemes that demonstrate this characteristic are often called *generic* or *functional* addressing mechanisms.

The line printer example could be solved by adding special software to the CMC for handling printers. This might be reasonable in a network populated mostly by printers, but it doesn't constitute a useful network model. A CMC could also be made to know about files and file servers, user terminal interfaces, and other specialized address and control spaces. One problem with this approach is that the CMC quickly becomes unmanageable - it is not possible to bound the size and complexity of the resulting program because there is always a need for "just one more" special accommodation. There are other problems such as possible deleterious effects new features may have on old features, space and time limitations as the number of 'features' and processes grow in a single program, and possible lack of a standardized CMC interface in a network consisting of many CMC's. What seems to be needed is a way of dealing with higher-level network control issues without complicating lower-level protocols and procedures.

The *manager* process mechanism solves higher-level addressing and control problems by indirection through the CMC. There are two parts: (1) means for addressing processes in the network by *name* rather than by network address, and (2) means for processes in the network to control virtual circuits in behalf of other processes (described in section 3.6). The line printer problem mentioned above would be solved by constructing a process in the network, the line printer manager, responsible for all line printers. The CMC then forwards requests for line printer service to the line printer manager which carries out the detail work of selecting a device and completing virtual circuit connections. In this scheme a line printer is an example of a network resource, network resources are owned by the manager processes, and are accessed only with the permission or with the help of the manager processes. In implementing this model a CMC must know about each addressible process in the network, but as long as sufficient connection management primitives are available to a manager process the CMC doesn't have to know what any manager process *does*.

The MGR primitives described below, together with the channel *splicing* primitives discussed in section 3.6, provide a starting point for network 'intelligence' based on process addressing. In this implementation manager processes are identified by an integer or by an ASCII string. Numbers are assigned by the maintainers of the network. The string-based mechanism is provided as an experimental alternative. Limits on string length are imposed by the present dependence on the 16-byte packet size supplied by the network. For this reason the string mechanism is considered to be temporary and not really available until packet size dependencies are removed from network control procedures.

3.5.1 T_MGR Primitive

A process becomes a manager in the network by sending (T_MGR,n)[-,-,-,-,-,-] to the CMC on a channel in call mode. This channel will be referred to as the manager's *cmc channel*, and is kept active by the manager during its lifetime. The number *n* is the manager's non-zero identification number. Only one instance of manager *n* is currently allowed within a single exchange. The CMC honors a T_MGR request with a standard response (see 3.1.3), or a P_LOG response in cases where the CMC requires a key to validate the 'identity' of the process claiming to be a manager.

The string interface for managers consists of sending the message (T_MGR,0)[*string*] to the CMC on a channel in call mode. The string name can be ten characters in length or may be null-terminated and less than ten characters. The T_MGR message will be acknowledged with the standard response by the CMC.

3.5.2 Manager Access Protocols

Managers are accessed through the CMC. That is, a user process sends a message identifying a manager process, and the CMC forwards the message to the manager process if possible. There are three message formats depending on whether the manager process is identified by an integer or string, and whether a virtual circuit should be set up. If the manager process is not available, the CMC sends a standard T_REPLY message back to the requester indicating failure. If the manager process is available, it is responsible for sending the T_REPLY message when new virtual circuits are set up. This arrangement is exactly analogous to the three-way message exchange between U, CMC, and L shown in Figure 3.2.1 if L is replaced by a manager process and T_SRV is replaced by T_MSRV.

3.5.3 T_MSRV Primitive

If a process sends (T_MSRV, *n*) [-,-,-,-,-,-] to the CMC on a channel in call mode where *n* is the identification number of a manager process, the CMC will establish a new virtual circuit to the manager if it exists. If the manager is not alive, the CMC sends a standard error response (see 3.1.3) back to the requesting process. If the manager is alive and well, the (T_MSRV, *n*) message is forwarded to the manager on the manager's cmc channel. In the forwarded message (T_MSRV, *n*) [-,-,-,-,-,-] is replaced by (T_MSRV, *ch*) [0,-,-,-,-,-] where *ch* is the channel number of the new virtual circuit at the manager's end of the connection. Note that this is similar to the listener's interface to the CMC which uses *param2* for the same purpose. The manager is responsible for sending a standard T_REPLY message back to the originator of the T_MSRV request on the new virtual circuit.

If the process sending (T_MSRV, *n*) is itself a manager process with id *id*, then the CMC forwards (T_MSRV, *ch*)[*id*, -,-,-,-,-] instead of (T_MSRV, *ch*)[0,-,-,-,-,-] as first described. A manager thus 'knows' when it is being accessed by another manager.

Once a manager and CMC are 'talking' on the manager's cmc channel, no other network process but the CMC can send to the manager on that channel. This permits a 'secure' method of authentication between managers because a (T_MSRV, *ch*)[*id*, -,-,-,-,-] can appear on a manager's cmc channel only if it is generated by the CMC. It is not possible for a non-manager process to forge a message and have it appear on another manager's cmc channel. That is, the method is as secure as the CMC implementation and its validation policies for manager authentication.

3.5.4 T_FSRV Primitive

If a process sends (T_FSRV,n) to the CMC on a channel in call mode, the message is forwarded to manager *n* if it exists. No virtual circuits are set up. No reply is sent to the requester. The forwarded message looks like (T_FSRV,n)[-,-, *sw*, *mod*, *ch*]. In the forwarded message *param3*, *param4* and *param5* are loaded by the CMC with values that identify the sender. The local switch number is in *param3*, module number in *param4*, and channel in *param5*. A manager process that receives T_FSRV can use the R_MAPTO primitive (see section 3.6.4) to set up a channel back to the module that originally sent T_FSRV.

This form of access is provided as a special case for downline loading microprocessors on the network. Since it might turn out to be useful for something else it was called T_FSRV instead of simply T_BOOT. The intent is to set up no channels at the behest of a microprocessor until it has been booted. This simplifies the boot rom code somewhat. More importantly it simplifies clean up procedures in the CMC when bootloading fails for some reason or when the boot load works but the program that was loaded doesn't work. With T_FSRV the channels used during boot loading are 'owned' by a manager process rather than booted processor.

3.5.5 T_SMGR Primitive

Access to string-named managers is via the message (T_SMGR, -)[*string*] where *string* is the manager's 'name.' If the manager can be accessed, (P_SMGR, *ch*) [*string*] is forwarded. A new virtual circuit is created and its number on the manager's end is passed along as *ch* as indicated in the message structure given above. As in the other cases the manager is responsible for sending a standard response on the new virtual circuit.

Since the string occupies most of the space in the message, the special processing described in section 3.5.3 that applies when a manager process contacts another manager cannot be implemented. This is one of several shortcomings of this implementation discussed in more detail below.

3.5.6 Commentary

The T_MSRV primitive should make the *param* fields in the message available to the process sending the message. Also the T_MSRV message forwarded to a listener by the CMC should have *param3* and *param4* filled in with the *arex* and *lad* of the caller as is done for the T_SRV messages forwarded to a listener. Before experiments with the string-named processes were carried out, the *param* fields were in fact treated as suggested above. However the crude implementation of the string facility destroyed *param* field processing for all T_MSRV messages.

Another problem with the manager mechanism as described is that there is no provision for addressing manager processes that are not on the local Datakit node of the caller. The simplest solution would be to permit network addresses in the T_MSRV message structure. A more interesting approach would be to let a T_MSRV message propagate in an appropriately limited way, say to the immediate neighboring nodes, to access a "non-local" manager process whenever a local one is not available.

Many improvements, some of them mentioned above, are needed in the current implementation. All the interesting and useful improvements depend on having larger messages containing one or more variable-length strings. We would like for *all* addressible processes to be named by a string rather numbering some of them. We would also like to pass file names, user identifications, and other items along in the T_MSRV message. In such an environment we would probably change the name T_MSRV to T_PSRV, designating the "connect to process" service.

It should be apparent that the manager process mechanism and its variants use the CMC as a software-implemented message switch. In this role the CMC directs T_MSRV messages to processes with a virtual circuit setup as a side-effect in some cases. This would still be the case for the various extended message structures imagined above - the contents of a T_PSRV or T_MSRV message would be translated to an address by the CMC, the entire message would be forwarded along with any additions. In this environment the message structures should probably be arranged with process name or function name near the beginning to simplify message parsing in the CMC.

Although the manager process implementation described here is a preliminary one, the next generation versions should have the same structure with some generalizations. This kind of higher-level addressing represents a significant improvement over 'classical' virtual circuit networks that provide limited address and translation capabilities.

3.6 Channel Map Primitives

3.6.0 General

The primitives described here are for setting up specialized virtual circuits (section 3.6.4) and for setting up 'ordinary' virtual circuits by a third party (3.6.1, 3.6.2, 3.6.3). The 'splice' primitives provide the third-party connection capability. They replace earlier attempts at indirect connection setup, but the goals are the same: third-party connection setups are essential to the development of higher-level services in a non-broadcast network like Datakit. The reasons behind this were brought out in section 1: it should be possible to introduce and modify services, i.e. manager processes, without touching the CMC. Therefore the CMC should provide the primitives needed to preserve the separation between manager processes and the CMC.

The sections that follow all describe the results of sending a T_MAP message to the CMC. The CMC will accept such commands on any channel in call mode. With the exception of R_MAPTO, the T_MAP primitives present something of a risk since a malfunctioning process could use the primitives to disrupt communications on random channels. Since the mapping primitives are provided for manager process use it may be necessary to restrict access in some appropriate way.

The protocol with the CMC is the same for each of the T_MAP primitives: a process sends a T_MAP message to the CMC and receives a T_REPLY response. Note that the syntax of the response is (T_REPLY, *code*) [*ch*, *err*, 0, *x*, 0,0]. This is the same as the 'standard' reply defined in section 3.1.3 with the addition of *x*. Each of the T_MAP operations defines param3 as a place where the requesting process can put its own code number *x*. This code is returned by the CMC in the corresponding T_REPLY message. This lets the process talking to the CMC multiplex the channel: several T_MAP operations can be requested without waiting for T_REPLY.

3.6.1 Splice Channels

Suppose a process C has an auxiliary channel to the CMC plus two other active channels to processes A and B as depicted in Figure 3.6.1. If the message (T_MAP,R_LSPLICE)[*c1*,*c2*,-,*x*,-,-] is sent to the CMC on the auxiliary channel, the CMC will establish a new virtual circuit between A and B, leaving two half-circuits to C, as depicted in Figure 3.6.2. The CMC will respond on the auxiliary channel with (T_REPLY,D_ACK)[-,-,-, *x*, -,-] if the splice is successful. The parameter *x* is not inspected by the CMC but is copied unchanged back to process C in the T_REPLY message. This helps process C multiplex the channel to the CMC if desired.

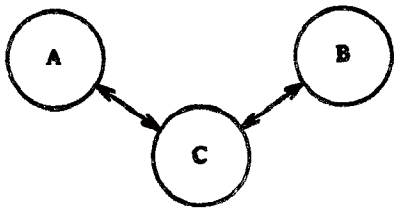

Figure 3.6.1

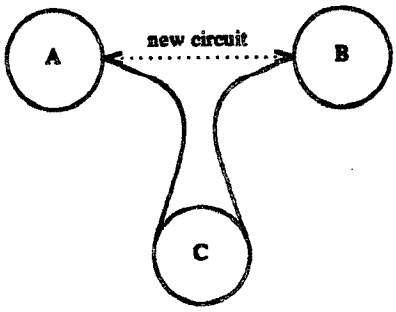

Figure 3.6.2

The two half-circuits will be removed by the CMC when process C or its listener executes a close handshake with the CMC (see section 3.3) for each channel. If process C is on Unix these handshakes are done when C closes its files for these channels. If C never opened the Unix file for a channel, the DIOCRESET *ioctl* call will make the listener carry out the desired close handshake with the CMC.

3.6.2 Splice Remote

The message (T_MAP,R_RSPLICE)[*sw1sw2, m1, c1, x, m2, c2*] performs a splice operation on two channels identified by the parameters. The value *x* is returned in the standard reply message as described earlier. The parameters spell out switch, module, and channel for each network address. The value *sw1sw2* is a 16-bit value with *sw1* as the low-order byte and *sw2* as the high-order byte. This primitive resembles R_LSPLICE in that it provides a kind of third party channel setup. The differences are that R_LSPLICE connects a pair of channels that terminate at the network address of the R_LSPLICE user whereas R_RSPLICE operates on a pair of channels that are 'remote' with respect to the R_RSPLICE user. This primitive is more 'physical' in nature than R_LSPLICE and its use is restricted: the command is accepted by the CMC only if the sender of the command is a network manager process (see 3.6) and there are listener or manager processes at each of the network addresses referenced in the message.

3.6.3 Splice

The message (T_MAP,R_SPLICE)[*cm1, c1*, 0, *x, cm2, c2*] also specifies splicing of channels at 'remote' network addresses. The primitive is less 'physical' than R_RSPLICE and represents more recent thinking on the subject of indirect connection setup. This primitive asks the CMC to set up a virtual circuit between two network addresses identified as the opposite ends of local channels *cm1* and *cm2*. The situation is depicted in Figure 3.6.3.

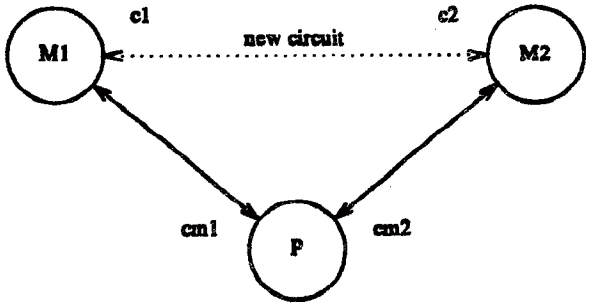

Figure 3.6.3

In the diagram P represents the process sending the R_SPLICE message, and M1 and M2 are the two modules identified by the channels *cm1* and *cm2*. The parameters *c1* and *c2* control how channels are selected for M1 and M2. *C1* controls the channel at M1, and C2 controls M2. If non-zero *c1* and *c2* specify particular channels. Either one or both of *c1* and *c2* may be zero. The CMC selects the channel for zero parameters.

The T_REPLY message from the CMC for the R_SPLICE primitive returns *c1* and *c2* in *param4* and *param5*. This is necessary because the CMC will choose channels on M1 or M2 if asked. The reply message is (T_REPLY,*code*)[*ch, err*, 0, *x, c1, c2*]. No messages are sent to M1 or M2 as part of the R_SPLICE protocol. It is assumed that the process P in Figure 3.6.3 notifies M1 and M2 over channels *cm1* and *cm2* when the new circuit is set up.

3.6.4 Map To

The message (T_MAP,R_MAPTO)[*sw*, *mod*, *chan*, *x*, -,-] may be sent to the CMC on any channel in call mode. It is interpreted as a request to create a virtual circuit using that channel to the hardware location specified by the switch, module, and channel address given as parameters. The CMC will honor such a request only if there does not exist a listener process for the specified network address and the address is not already in use. This mechanism is intended for constructing diagnostic and maintenance software. The restrictions on its use preclude interference with other network operations.

3.7 Message Summaries

to CMC

type	*srv*	*param0*	*param1*	*param2*	*param3*	*param4*	*param5*
T_LSTNR	0	lhn	ack	reset	0	0	0
T_SRV	service	arex	lad	mode	-	-	-
T_FSRV	manager						
T_CMC	0	arex	ack	reset	link	-	-
T_REPLY	reply	lch	msg	rch	x	-	-
T_CHG	D_CLOSE	-	chan	-	-	-	
T_CHG	D_ISCLOSED	-	chan	-	-	-	
T_MAINT	M_NAIL	swtag	m1	c1	m2	c2	-
T_MAINT	M_PRY	sw	m1	c1	-	-	-
T_MAINT	M_CONFIG	sw	mod	mtype	nchan	-	-
T_NMS	M_LIVE	-	-	-	-	-	-
T_MAP	R_LSPLICE	c1	c2	-	x	-	-
T_MAP	R_MAPTO	sw	mod	chan	-	-	-
T_MAP	R_RSPLICE	sw1sw2	m1	c1	x	m2	c2
T_MAP	R_SPLICE	cm1	c1	0	x	cm2	c2
T_MGR	n	-	-	-	-	-	-
T_MGR	0	*string*					
T_LOC	D_TIMER	-	-	-	-	-	

to L

T_SRV	service	arex	lad	n	arex	host
T_CHG	D_CLOSE	-	n	-	-	-
T_CHG	D_ISCLOSED	-	n	-	-	-
T_LOC	D_CLOSE	n	-	-	-	-
T_LOC	D_TIMER	-	-	-	-	-

3.8 Implementation Techniques

The following sections present code fragments and state tables for those parts of a user process, the CMC, and listener processes that participate in ordinary virtual circuit setup and takedown. The C structures used below were introduced in section 3.1 and need not be reproduced here.

3.8.1 Connection Setup

The *connect* subroutine shown here may be taken as a prototype for all user program interfaces with the CMC. It formats a *dialout* message, sends it to the CMC and waits for a reply. The explicit conversion from local to network data representation is done by the routine *dktcanon*, which copies from *msg* to *buf* with translation controlled by the string argument. Conversion from *buf* back to *msg* is done by *dkfcanon*. The *select_channel*, *send*, and *set_alarm* routines are generic names for functions that would be implemented on UNIX with *open*, *write*, and *alarm* system calls.

Note that the *connect* routine only gets a virtual circuit. Each service requires that the circuit be 'conditioned' by some transport protocol, and most services require an authentication exchange, or login, before service can be provided. These exigencies could be dealt with by the code that calls on *connect*, or the appropriate system-dependent mechanisms for turning on protocols and establishing identification could be added to the code for *connect*. The *dkdial* routine described in section 6 takes a hybrid approach: it does the function of *connect* plus the login funtion, avoiding the manipulation of transport protocols. The point is that the network control architecture treats virtual circuits, authentication, and transport as *separate* issues in a way that permits the software to deal with them separately or in a combined way as circumstances and network evolution dictate.

```
/*
 * Connect to server at network address (arex,lad).
 */
connect(arex, lad, service)
{
struct dialout msg;
short buf[sizeof msg];
int cc, rcc;

     ch = select_channel();          /* get outgoing channel */
     if (ch==ERROR)                  /* return on error */
          return(FAIL);
     msg.type = T_SRV;               /* prepare dialout message */
     msg.srv = service;
     msg.param0 = arex;
     msg.param1 = lad;
     msg.param2 = 0;
     msg.param3 = 0;
     msg.param4 = 0;
     msg.param5 = 0;

     cc = dktcanon("cchhhhhh",&msg,buf);  /* convert representation */
     send(ch,buf,cc);                     /* send msg */

     set_alarm(15);                  /* set timer */
     rcc = read(ch, buf, cc);        /* wait for reply */
     set_alarm(0);                   /* turn off alarm */
     /* On Unix rcc == -1 if the alarm goes off. */
     if (rcc==cc) {
          dkfcanon("cchhhhhh",buf,&msg);
          if (msg.type==T_REPLY && msg.srv==D_OPEN)
               return(ch);
     }

     release(ch);
     return(FAIL);
}
```

3.8.2 CMC Channel Control

This section presents one of many possible state machine representations of the CMC's part of connection setup and takedown as well as a code segment indicating one way of translating the protocol into working code.

It is convenient to represent the operation of the CMC as a composition of "channel machines" or simple state machines. This is done below where two machines are given: the *A machine* for the channel to a user process and listener, and the *B machine* for the channel to a server process and its listener. Each machine has three states and responds to five input messages. The output messages of each machine are represented syntactically by P!M, where P is a letter representing a process and M denotes a message. The values for P are U, S, A, and B representing the user, server, A machine, and B machine processes. Figure 3.8.1 illustrates the relationships between the processes.

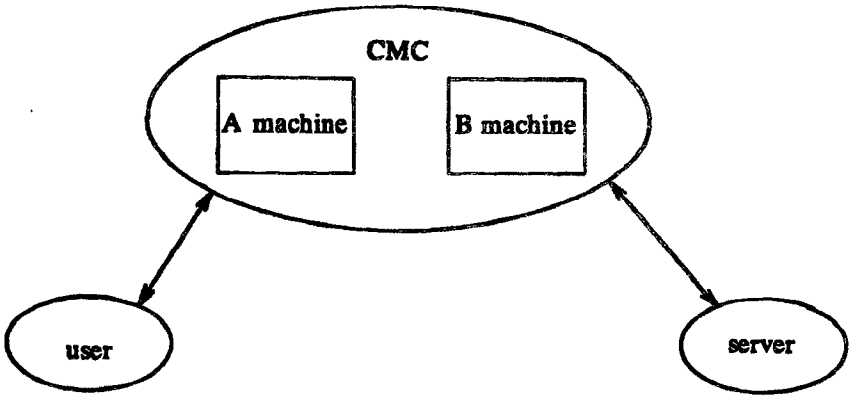

Figure 3.8.1

In the table the T_SRV, CLOSE, and CLOSED messages are a shorthand representation for the CMC messages that were defined in section 3.2. The DOWN message is sent between A and B machines as notification that the channel is being taken down. The TIMER message denotes the CMC's internal 15-second alarm clock event. Lastly, the *setup* and *release* actions in the tables represent the making and breaking of virtual circuits in the network. A blank entry in the *next_state* table entries below means that the next state is the same as the current state.

A channel machine

inputs	T_SRV	CLOSE	CLOSED	DOWN	TIMER
state	*next_state* :*action(s)*				
IDLE	OPEN :setup chan :B!OPEN	:U!CLOSED			
OPEN	WAIT :U!CLOSE :release chan :B!DOWN	IDLE :U!CLOSED :release chan :B!DOWN		WAIT :U!CLOSE	
WAIT	OPEN :B!OPEN :setup chan	IDLE :U!CLOSED	IDLE	:U!CLOSE	WAIT :U!CLOSE

B channel machine

inputs	OPEN	CLOSE	CLOSED	DOWN	TIMER
state	*next_state* :*action(s)*				
IDLE	OPEN :S!T_SRV	IDLE :S!CLOSED			
OPEN	WAIT :S!CLOSE :release chan :A!DOWN	IDLE :S!CLOSED :release chan :A!DOWN		WAIT :S!CLOSE	
WAIT	OPEN :S!T_SRV	IDLE :S!CLOSED	IDLE	:S!CLOSE	WAIT :S!CLOSE

Each channel starts off in the IDLE state. Reception of D_CLOSE on any IDLE channel will result in a D_ISCLOSED reply with no change in state. A T_SRV message starts the connection setup procedure. The A machine passes the connection setup to the B machine as indicated by B!OPEN. The B machine forwards the service request to the server with S!T_SRV. Both CMC state machines remain in the OPEN state until one or the other or both sides releases the channel. The channel takedown procedure is started by reception of CLOSE by one or both of the state machines. A machine seeing a CLOSE event sends the CLOSED acknowledgement and notifies the other machine with a DOWN message that it should take down the 'other' end of the connection.

3.8.3 Listener

The listener's state machine is given below. It is formulated like the machines presented for the CMC part of the protocol. The two message types, L.open and L.close signify events in the listener that accompany channel open or channel close operations by other processes on the listener's machine.

The error cases in this machine detect inconsistencies in the listener's environment and the network control. The *error1* states signify illegal channel reuse by the network; i.e. the network control software sets up a virtual circuit to a channel that is supposedly already set up. This really isn't supposed to happen, but if it does the listener has little choice but to abort the process currently using the channel. The listener may then choose to accept or reject the incoming T_SRV message. The *error2* states signify errors in the listener's software environment: appearance of an L.close without a corresponding L.open, and L.open of a channel that is being closed.

Listener machine

inputs	T_SRV	D_CLOSE	D_ISCLOSED	L.open	L.close	TIMER
state	*next_state* :*action(s)*					
IDLE	OPEN :T_REPLY	IDLE :ISCLOSED		OPEN	error2	
OPEN	error1	WAIT2 :hangup			WAIT1 :CLOSE	
WAIT1	OPEN :T_REPLY	IDLE :ISCLOSED	IDLE	error2	error2	WAIT1 :CLOSE
WAIT2	error1	WAIT2	WAIT2	error2	IDLE :CLOSE	WAIT2 :hangup

4. Servers

4.0 General

This section documents the message interface to some existing network servers. If the network address (3.1.2) of a server or service is known, access to the entity is obtained using (T_SRV,*code*)[*arex*, *host*, *mode*, -,-,-] to set up the virtual circuit as described in 3.2.2.

4.1 Autologin

The Unix-based listener program requires the identity of the user before starting up most network services. This could be obtained by forcing interactive users through a login procedure before starting any server process. This would be impractical for non-interactive network access where programs are not run directly from interactive terminals. Forcing a login for every command is inhospitable in any case. The so-called *autologin* procedure is a compromise method that allows the *dkdial* routine (see section 6) to automatically log in at a remote host.

The procedure is as follows:

1. After receiving a standard T_REPLY message (see sections 3.1.3, 3.2) indicating that a new virtual circuit has been set up, the calling process sends the ASCII string representation of the user's login name on the new virtual circuit. The string must be null-terminated and in the present implementation must fit within a 16-byte Datakit packet. The remote listener waits for this string to arrive on the new virtual circuit after sending the T_REPLY but before providing service.

2. If the string is accepted by the listener, the listener responds with (T_REPLY, D_OPEN). and starts up the requested service on the new channel.

3. If the string is not accepted by the listener, the response is (T_REPLY,D_PLOG) which stands for "please log in." At this point the calling channel is handed over to a normal login procedure. The network service is started only after the login is successful.

The listener program contains within it a list of network exchange addresses for which the local system wishes to allow autologins. If a T_SRV request comes in from an exchange or machine not on the list, the formal login/password exchange initiated by D_PLOG will be forced. Also, autologin for privileged user ID's is disallowed.

The autologin procedure is a compromise as already mentioned. It requires that one have an account on each of the machines where autologins are wanted, but it does not require that the integer account *numbers* be the same. This is a desirable property. Since the CMC is responsible for filling in the network address of a caller, the exchange and logical address numbers can be 'trusted' as a basis for discriminating between callers. This is another good property. One drawback of the original implementation was that there was nothing to prevent a user from making a private version of *dkdial* that would log its owner in as anyone, except privileged users, on a remote system. This problem was recently addressed by Andrew Koenig who modified the device drivers and *dkdial* routines so that (1) the driver restricts the call setup operation to privileged processes, and (2) the *dkdial* routine runs a 'safe' privileged process that does the actual setup protocol and login handshake. This is, of course, just the tip of the security iceberg. The improved *dkdial* can be compromised by any user who can break the normal access checks on his machine. Lastly, no matter what steps are supposedly taken in the software of a computer making a network connection, there is no general way for a computer receiving a connection setup message to verify that the software on the calling machine was not compromised.

4.2 T_SRV Servers

The services accessible via the *srv* field of a T_SRV message are outlined below. Services are named by their *srv* code as it appears in */usr/include/dk.h*. Although the terminology used to describe the services comes from Unix, translations to other system environments should be obvious.

D_XSH - perform autologin, turn on a stop/start flow control, turn off echo, run the shell. This is the original virtual terminal interface and is seldom used.

D_YSH - perform autologin, turn on the so-called TDK flow control discipline, run the shell.

D_SH - carry out autologin, turn on the trailer protocol, run the shell.

D_LOG - turn on the TDK flow control, and run the system *login* procedure. This is the present interface between terminal controllers and hosts.

D_FS - perform autologin, turn on the packet driver protocol (see section 5), run the file server program.

D_XFS - perform autologin, turn on the packet driver, run a test version of the file server.

D_EXEC - perform autologin, turn on the packet driver, read a null-terminated string, execute the string as a Unix command.

D_TREXEC - perform autologin, turn on the trailer protocol, proceed as with D_EXEC.

D_NULL - do nothing; the channel is closed if not in use by some process within 15 seconds.

The multiplicity of ways to connect to a Unix shell, D_XSH, D_YSH, and D_SH, and to execute a remote process are evidence of the random growth of conventions as network software evolved. The technical reason for the profusion of codes is that it was 'easer' at one time to expand in the direction of *srv* codes than to work out a proper mechanism for protocol selection. This would have required reworking some message formats. What you see above is the result of taking the easy way out. Although synonyms for the same service will probably be eliminated in future editions of the software, it will be done at some expense - much more so than the relative cost of making it cleaner to begin with. It is hoped that this small example will be of some benefit to network architects who might be tempted by 'interim' solutions to network control problems.

The D_EXEC operation is the basis for most of the networking programs that have been implemented: bulk file transfer, file archiving, and remote execution of many commands. Except for the differences in transport protocols, D_EXEC could subsume the other services listed above. Some progress in this direction has already been made. Dennis Ritchie recently adjusted the Unix login program, the one referred to with D_LOG above, so that instead of always running the shell as a default the program will accept a string to be executed. With this improvement the listener program always runs *login* which always runs the desired server or process. If messages exchanged between listeners and network control programs could contain arbitrary length strings, then the autologin information plus the string to be executed could be passed to a listener as one message. This would eliminate the various message exchanges described above and unify the separate access procedures for servers.

5. Trailer Protocol

5.0 General

Several different transport protocols are in use on Datakit networks. They vary greatly in complexity, performance, and function. The trailer protocol results from experience with several of these, including the *packet driver* protocol, [1] and from efforts to accomodate properties of Datakit hardware and UNIX-based software into protocol design.

The trailer protocol is a procedure for end-to-end error control, flow control, and out-of-band signaling between a pair of processes communicating over a virtual circuit. Error control provides for retransmission of lost data. Flow control regulates the speed of transmission to the receiver's input processing rate. Out-of-band signaling allows user processes to mark and also to interpolate command information into byte streams.

The protocol takes as the definition of virtual circuit the one provided by Datakit: a virtual circuit is like a transmission line in that it may damage or lose information but will deliver messages in the same order as they are sent. It is also assumed that the parity checks built into the Datakit hardware and device handlers are adequate for detecting transmission errors. Since the network discards any data whose corruption is detected by parity and cyclic redundancy checks in the hardware, transmission errors appear at the protocol layer as an absence of data.

5.1 Trailer Protocol

The trailer protocol is so-called because the message format consists of a sequence of zero or more data bytes terminated by a trailing control message. The boundary between control and data is marked by a control byte (see section 2.1). Several versions of this protocol have been tested in the past. The first implementations constrained the control byte to be the first byte of a hardware packet. That is, the trailers were represented as control packets as defined in section 2.1. This restriction was imposed by the Datakit device driver and not by the protocol and is mentioned for historical interest. No such restrictions are to be understood here - the protocol defines a byte stream algorithm independent of hardware framing.

The technique of using trailers instead of headers as in other transport protocols has certain merits: a transmitter can begin sending a message without knowing how long it will be - a trailer will eventually terminate the message. In addition buffer management and error control in the receiver are less complex than for header-based protocols. Because a response to a received message cannot be generated correctly until the entire message has been read, sequence numbers and bytes counts at the beginning of a message must be stored as the message arrives. This creates buffer management problems. By using trailers instead of headers, control information becomes available precisely when it is needed.

5.2 Message Formats

The messages in this protocol consist of a (possibly null) data part followed by a fixed length trailer. The trailer format consists of an 8-bit type code followed by an 8-bit sequence number, a 16-bit parameter, and a terminating control byte. The table below enumerates the messages – capitalized names represent constants defined in */usr/include/tr.h*, '-' indicates padding, <data> denotes zero or more data bytes, and <> denotes zero data bytes.

The *emark* symbol in the table is the *end of message* control character. In normal operation the receiver stores incoming data until an *emark* byte is seen. If input buffers overflow before this happens, the saved data is discarded.

A message may be terminated with either of *emark* as explained above or *ebad*. A message ending with *ebad* is to be discarded. This rejection code could be generated by the sender of the message if it decides to abort the transfer. It might be generated by hardware or software at the

[1] The packet driver is documented in TM-11273-81-4.

receiver while checking the hardware parity indicators or the contents of a trailer. That is a lower-level layer may translate *emark* into *ebad*.

TR Protocol Messages

data	*type*	*sequence*	*param*	
<data>	P_DATA	xseq	nbytes	*emark*
<data>	P_CMD	xseq	nbytes	*emark*
<data>	P_MORE	xseq	nbytes	*emark*
<data>	P_ERROR	xseq	nbytes	*emark*
<>	P_REJ	rseq	-	*emark*
<>	P_ALLOC	rseq	nbytes	*emark*
<>	P_RALLOC	rseq	nbytes	*emark*
<>	P_SYNC	seq	size	*emark*
<>	P_RESYNC	-	-	*emark*
<>	P_ENQ	eseq	epoch	*emark*
<>	P_RESP	eseq	epoch	*emark*

The P_REJ, and P_ALLOC messages may appear within *<data>* segments. Although no implementations at present generate transparent control messages, it is expected that future implementations will. Therefore receivers must be prepared to recognize these messages without error if they happen to be embedded in data segments.

The definition of the type control characters includes a bit P_CHK that is set for each of the transport types and cleared for all the others. This distinguishes the transport messages as a class and simplifies an implementation since only transport messages have sequence numbers checked for monotonicity and have non-null data segments.

The multiplicity of transport messages deserves some explanation. P_DATA is the principal carrier of data. The P_MORE message is used as a sub-record delimiter when records larger than *bsize* (bsize defined below in 5.3.0) are to be passed through the protocol. A sequence of P_MORE messages is terminated by a P_DATA message. The P_ERROR message provides an out-of-band error message path (5.3.4). The P_CMD message operates like the P_DATA message, but signifies that the contents of <data> is command or control information in a higher-level protocol as distinguished from data in that protocol.

5.3 Informal Description

5.3.0 Outline

This section introduces terminology and briefly describes the major parts of the protocol.

The words 'reader' and 'writer' will be used to distinguish between application programs using an implementation of the protocol and 'receiver' and 'sender' which denote components of an implementation. For example data flows from writer to sender to the network and thence to receiver and finally a reader. The adjective *opposite* will be used to denote entities at different ends of a virtual circuit. For example a sender transmits data to the opposite receiver. Similarly *local* denotes entities on the same end of a virtual circuit. For example a receiver may cause its local sender to transmit acknowledgement messages to the opposite sender.

The protocol is defined between four entities: two senders and two receivers. Each sender maintains an 8-bit counter *xseq* corresponding to the number modulo 256 of messages transmitted over the virtual circuit, another 8-bit counter *lseq* which is the sequence number of the most recently acknowledged message, a value *avbytes* (controlled by the opposite receiver) which is the number of bytes that may be sent, and a value *xbytes* which is the number of bytes that have been sent but not acknowledged by the opposite receiver. Each sender also observes a parameter *bsize* supplied during initialization by the opposite receiver. *Bsize* is the maximum number of data bytes

allowed in any single message.

The sequence numbers in messages are intended for error control only - not flow control. Sequence numbers are needed to detect missing messages in an otherwise correct stream. The sequence number space is made large enough so that the byte counts used for flow control will normally control the flow of data long before the sequence numbers wrap around. Of course a test for sequence numbers wrapping is needed for correctness, but this turns out to be a test for equality of two numbers. This test is much simpler than the arithmetic needed to do "sliding window" calculations as required by typical protocols with 3-bit sequence numbers spaces that use the sequence numbers for both error control and flow control. This may seem like a minor issue, but it turns out to have noticeable consequences when one begins to implement a transport protocol in special-purpose hardware.

Each receiver maintains a counter, *rseq*, that holds the sequence number of the most recent correctly received message. Each receiver manages a buffer area of some persuasion where the total number of bytes the receiver can commit to the circuit is denoted by *nalloc*. The maximum number of outstanding sequence numbers a transmitter is allowed to have, normally called a *window*, is 255. The minimum value for *nalloc* is 128 bytes. The maximum value is 16384. *Bsize* is constrained by an encoding scheme to be *nalloc/i* where *i* may be 1, 2, 3, or 4.

The error control algorithm in the trailer protocol uses an adaptive timeout mechanism. This lets the protocol implementation adjust to variations in propagation delay and system loading. It is assumed that a system clock *ttime* is available that increments at least every 1/60 second. Each sender records in a variable called *epoch* the value of *ttime* for the most recently transmitted message. Each sender also maintains a variable called *wtime* which is the error control timeout delay, i.e. the length of time in *ttime* ticks to wait without a reply from the opposite receiver. *Wtime* is initialized to 1 second and adjusted up or down by the sender according to observed delays. Each sender also maintains a counter called *eseq* that is incremented each time the sender enters a blocked state because of either flow control or error control. This counter is sent in P_ENQ messages and echoed by the opposite receiver in P_RESP messages so that a sender can put the communication link in a known state and measure the end-around delay. The timeout control algorithm has some hysteresis - it is biased more toward increasing timeout delays rather than decreasing them. This assures that the timeout controls will not oscillate when faced with stochastic network and operating system delays.

The messages in this protocol may be classified in the following way: *transport* messages are represented by P_DATA, P_CMD, P_MORE, and P_ERROR; *flow control* messages are represented by P_REJ, P_ALLOC, and P_RALLOC; *control* or *handshake* messages are represented by the following command/response pairs - P_SYNC/P_RESYNC, and P_ENQ/P_RESP.

The transport messages share the same sequence number space. The flow control messages are sent in response to received transport messages. The control messages operate in pairs; e.g. the response to P_SYNC is P_RESYNC.

The variables and parameters described above define the state of a communication link as modeled by this protocol. These are in summary *xseq, lseq, avbytes, xbytes, bsize, epoch,* and *wtime* for a sender and *rseq, nalloc,* and *bsize* for a receiver.

5.3.1 Initialization

Initialization is accomplished with a pair of two-way handshakes: P_SYNC/P_RESYNC. Each sender transmits P_SYNC messages repeatedly. When SYNC is received, RESYNC is sent in return. When a receiver has detected both SYNC and RESYNC or has observed both SYNC and one of the transport messages, it then concludes the channel is live.

Each receiver chooses the initial values for *rseq, nalloc,* and *bsize*. These are sent in P_SYNC messages to the opposite sender. *Nalloc* and *bsize* are encoded in the *size* field of the P_SYNC message. The lower 14-bits are for *nalloc*. The top two bits *xy* select a value for *bsize*. If *xy* is zero, then *bsize* is *nalloc/1*. If *xy* is one, then *bsize* is *nalloc/2*. In general *bsize* is *nalloc/(xy+1)*.

The semantics of P_SYNC are that the local *xseq*, *lseq*, *bsize*, and *avbytes* are initialized by the values obtained from a received P_SYNC message. *Xseq* and *lseq* are loaded from the *seq* field. *Bsize* and *avbytes* are extracted from the *size* field as explained above. *Xbytes* is initialized to zero.

The initialization messages may be repeated after the protocol has been transporting data. This is called a *secondary initialization*. The purpose is not to renegotiate the parameters for a session although it could be. Instead secondary initialization is a 'feature' used to avoid the overhead of tearing down and then recreating the software context of a protocol when we want to replace one end of a virtual circuit by an alternate destination. This situation arises when a terminal or process that is at one moment connected to machine A desires to switch to machine B. The problem to be solved is that of getting the sequence numbers and other state variables to agree when the switch is made to B.

Either side may begin a secondary initialization by sending a P_SYNC message. When P_SYNC is received the proper action is to duplicate the handshakes described above: respond with P_RESYNC and send P_SYNC messages until P_RESYNC is returned. In examples where secondary initialization is useful senders tend to be quiescent before and during the procedure, i.e. they have no data to send. In the event that a secondary initialization begins and a sender is not quiescent, any output transmissions should use the 'new' sequence numbers starting with *seq* from the P_SYNC message. This may require renumbering some output messages.

5.3.2 Data Transport

5.3.2.0 Output Control

Values of *xseq*, *lseq*, *xbytes*, and *avbytes* are established by a P_SYNC/P_RESYNC exchange for both senders. Each time a transport message is to be sent *xseq* of the sender is incremented to become the sequence number of the new message. However if the value of *xseq* would become equal to that of *lseq*, then the increment is not done. The message may be sent only if the increment operation is actually carried out *and* if the result of subtracting both *xbytes* and the size of <data> in the new message from *avbytes* is non-negative. When a message is sent the value of *xbytes* is incremented by the size of <data> before any additional output processing.

Note that the test for *xseq* equaling *lseq* is the sequence number wraparound test, not a window flow-control mechanism. The real flow control is done by the byte count tests.

Note also that the size of <data> is allowed to be zero. A receiver must faithfully reproduce zero-length writes through the channel by delivering a corresponding number of zero-length messages to the reader. The propagation of zero-length messages is a simple mechanism for placing marks in the end-to-end byte stream. This is discussed in more detail in section 5.3.4.

The conditional increment operation on *xseq* prevents the sequence number 'window' from wrapping around. Without such a test the sequence numbers would not map uniquely to outstanding messages. The large (256) sequence number space was chosen for three reasons: (1) 'large' numbers of small messages can be handled, (2) flow control can be sufficiently maintained via byte counts alone, byte counts plus complicated sequence number checks are redundant and incur more running overhead, (3) sequence number management on hardware with 8-bit arithmetic can be done directly without bit masks and modulo arithmetic.

5.3.2.1 Receiver Error Control

A receiver checks that incoming transport messages are properly framed, checks incoming sequence numbers for monotonicity, and checks the value of *nbytes* in the trailer against the number of bytes (sizeof <data>) actually received. If any of these tests fail the message is discarded. Upon correct reception of a transport message a receiver increments *rseq*, which should then be the same as the received *xseq*, and responds immediately with P_ALLOC. This message acknowledges correct reception of message number *rseq* and also transmits a current value for *avbytes* to the sender.

A receiver enters the error state when one of the trailer checks mentioned above fails or when its input buffer area overflows. In the error state the receiver does nothing to *rseq*, does not send P_ALLOC, and may send P_REJ. The P_REJ message uses the current (unmodified) value of *rseq*. A P_REJ message thus contains the sequence number of the last correctly received message. This informs the transmitter that all messages with sequence numbers up to and including *rseq* were successfully transmitted and that an error was detected after the last good message. The transmitter should begin retransmitting messages starting after the sequence number received in an P_REJ message.

After sending one P_REJ message a receiver may *not* send another without having first received a correct message (and thus sent P_ALLOC). This restriction makes P_REJ equivalent to a fast transmitter timeout and avoids the multiple retransmission problem that arises when the first of a sequence of messages is received in error.

Under certain conditions a sender may transmit a P_ENQ message (sections 5.3.2.3 and 5.3.3). The receiver responds to this by sending a P_RESP message followed by a P_RALLOC message. The *eseq* and *epoch* parameters in the P_RESP message are copied from the P_ENQ message. That is, the P_RESP message echoes the data from the P_ENQ message. A P_RALLOC message is formed exactly like a P_ALLOC message: it sends the receiver's current value for *rseq* and buffer space. However the *rseq* field in the P_RALLOC message is interpreted differently from that in the P_ALLOC message (see 5.3.2.3).

5.3.2.2 Receiver Flow Control

A receiver generates P_ALLOC messages when trailers are received as explained above. In addition P_ALLOC messages are generated when data is transferred from receiver to reader. Each P_ALLOC contains the current value of *rseq* and a byte count. From the receiver's point of view this byte count is the number of additional bytes it is prepared to accept from the sender. From the opposite sender's point of view this byte count is used to calculate a new value for *avbytes* (explained in 5.3.2.3).

Let R represent the amount of acknowledged data in a receiver's buffer area that is waiting to be delivered to a reader. Let S represent the additional buffer space available to the receiver for more data: i.e. S = *nalloc* − R. S is the value sent in P_ALLOC and P_RALLOC messages. We assume that *nalloc* remains constant for a receiver during a session although this is not strictly necessary.

In this flow control method S decreases as data arrives at a receiver and is acknowledged. Additional buffer space becomes available as the received data is transfered to a reader, and S therefore increases. In general P_ALLOC messages are generated when S changes.

Some optimizations are available to the receiver. It is left up to the an implementor whether or not to abstain from sending P_ALLOC messages every time that S increases. For example if the reader is picking up bytes one at a time, the receiver may elect to send P_ALLOC only after the increase in S exceeds some threshold. On the other hand no such liberty may be taken with the P_RALLOC message: it should represent the true state of a receiver.

5.3.2.3 Sender Flow Control

When a P_ALLOC is received, the local sender releases message buffers for messages with sequence numbers *lseq* +1 through the value of *rseq* in the P_ALLOC message. The new value for *lseq* is *rseq*. *Xbytes* is decremented by the number of bytes in the just-released message buffers. *Avbytes* takes on the value *nbytes* from the P_ALLOC messages.

The *nbytes* value in a P_ALLOC message is an 'absolute' rather than 'relative' value. This means that if a pair of identical P_ALLOC messages were received in the presence of no other activity, the second one would have no effect on the state variables. This also means that a receiver can send P_ALLOC with a byte count of zero to halt the byte stream.

If a sender is blocked, waiting on *avbytes* to increase, and deduces (by timing out) that P_REJ, or P_ALLOC are not forthcoming, then P_ENQ/P_RESP may be used to determine the true state of

affairs. The blocked sender may transmit P_ENQ and expect to receive P_RESP in response. Since the receiver is supposed to send P_RALLOC after sending P_RESP, the data flow should start anew with one difference: if the *rseq* field in the returned P_RALLOC message differs from the sender's value for *xseq*, then the message has the same effect as P_REJ and retransmission should begin at the sequence number following *rseq*.

5.3.3 Timeouts

There is but one timeout *wtime* defined in the protocol. It is used by the sender, and its purpose is to detect transmission errors by noticing that output messages have not been acknowledged by the opposite receiver. The procedure to be followed when the timeout period expires is as follows:

1. increment *eseq*.
2. send P_ENQ messages until there is a P_RESP response. The local value of *ttime* is sent with each P_ENQ message in the *epoch* field. Returned P_RESP messages are discarded until one with a matching *eseq* field is returned.
3. The difference between the value in a returned *epoch* field and the local *ttime* approximates the round-trip message delay.

If the round trip delay calculated by a P_ENQ/P_RESP exchange is greater than *wtime* then the observed delay is used as the new value for *wtime*. This increases the timeout in response to network delays or operating system delays at the opposite end - it is not practical to differentiate between the two.

The value of *wtime* is decreased in two circumstances:

1. if the round-trip delay measured by a P_ENQ/P_RESP handshake is less than 1/2 the existing value for *wtime*, *wtime* is halved;
2. if a sender blocks on flow control but becomes unblocked before *wtime* expires, then the amount of time spenting waiting becomes the new value for *wtime*.

These conditions for decreasing *wtime* are somewhat conservative, providing needed hysteresis in the system. It is important that timeout periods increase more quickly than they decrease. The policy described here could be made even more conservative if experience proved it necessary. The most direct method would be to require that the two timeout-decreasing circumstances hold for a time interval that exceeds some multiple of *wtime* time units. Arbitrary smoothing of timeout response characteristics could be introduced in this way.

5.3.4 Out of band signals

The trailer protocol provides out of band signalling in three ways: zero length records, the P_CMD message and the P_ERROR message.

The zero propagation mechanism can be used in many instances to simplify higher-level protocols. For example a stream-oriented bulk file transfer mechanism has been built that sends a file in two parts, each part terminated by a zero-length record. The first part or header consists of the file name plus ownership, and mode. The second part is the file. It is easy for the file transfer reader to switch between header processing and file copying after observing the terminating zero-length records. Other applications involving the use of existing Unix software need the zero-length record facilty or some equivalent since a zero-length read is the standard indication of the end-of-file condition for the system.

It is of historical interest to note that the original packet driver (see earlier footnote) provided zero propagation. The uucp program which contains the packet driver utilizes zero propagation to mark the ends of files, although the exact technique is more complex than the one described above.

The P_ERROR message provides a separate 'channel' for multiplexing error messages from higher level processes to a terminal. The intent is to provide a place to put Unix *stderr* messages.

The P_CMD message provides a way of interpolating command and data information. For example a virtual terminal protocol may implement a local echo facility instead of remote echo. If the terminal protocol is to be transparent to Unix application programs, then program-generated

commands that change the state or mode of the local terminal must be propagated across the virtual circuit. This situation is solved in the trailer protocol by sending data to be printed or displayed in P_DATA messages and mode commands in P_CMD messages.

5.3.5 Shutdown

Earlier versions of this protocol defined an explicit closing handshake, but this is not strictly necessary. Instead either side may "clean up" the link by sending a zero-length message in the normal manner and then fall back on the channel shutdown procedures (section 3.3) to complete the job.

6. LIBDK Routines

6.0 Introduction

Most of the network access protocols (section 3) that are meant for user programs have been encapsulated as library routines. On Unix these routines are kept in the file */lib/libdk.a*. They are summarized below.

6.1 Channel Setup Routines

```
struct dkaddr
netname(name)          - return network address for name.

dkdial(srv,dkaddr)     - connect to service srv at network address lad, arex.
struct dkaddr dkaddr;

dkcall(srv, name)      - connect to service srv on host name.
char *name;

dkexec(name, cmd)      - execute the shell command line cmd on host "name".
char *name, *cmd;

dkaux()                - obtain an auxiliary channel.

dkmgr(id)              - become manager process with id id.

dkmap(sw, mod, ch)     - obtain channel to a physical address.
```

The *netname* routine returns the network address corresponding to the string *name* in the form of a *dkaddr* structure which has the form:

```
struct dkaddr {
    short area;
    short exch;
    short lad;
};
```

This routine always returns a structure value. Errors are indicated by returning a value of -1 for *lad*. Note that this routine preempts the value -1 for a network logical address.

The subroutines *dkdial* through *dkmap* return a Unix file descriptor if successful or -1. Each of these routines is meant to establish a virtual circuit in the network. The virtual circuits are accessed by read/write operations on the returned file descriptors.

Dkdial executes the T_SRV channel setup discipline to return a file descriptor to service *srv* at network address *host* plus *arex*. The *dkcall* routine has the same general purpose as *dkdial*, but translates *name* to a network address. Both *dkcall* and *dkdial* set a global variable *dkrchan* with the channel number obtained at the remote host (see *param2* in 3.2.2).

The *dkexec* routine returns a file descriptor connected to the standard input and output of a process executing command *cmd* on the computer specified by *name*.

Maintenance and other programs can use *dkaux* to obtain a file descriptor for an auxiliary channel (3.4.0) to the CMC.

A manager process with identification number *id* makes itself known to the CMC by using *dkmgr(id)* which returns a file descriptor for a channel to the CMC.

Dkmap returns a file descriptor for channel *ch* on module *mod* on switch *sw* where *sw* is interpreted as a particular hardware switch under jurisdiction of the local CMC. That is *dkmap* is not meant to work across exchange codes.

6.2 Channel Manipulation

`dksplice(dk, cm1, c1, cm2, c2, x)` - splice two channels using R_RSPLICE.

`dklsplice(dk, c1, c2)` - splice two channels with R_LSPLICE.

`dkminor(dk)` - return channel number for file descriptor *dk*.

If *dk* is a file descriptor for an auxiliary channel to the CMC, then *dksplice* and *dklsplice* will splice together channels according to the description in section 3.5.

If *dk* is a file descriptor for a Datakit channel, *dkminor(dk)* returns the channel number.

6.3 Data Conversion

`dkfcanon(fmt,from,to)` - convert canonical to local format.
```
char *fmt,*from,*to;
```

`dktcanon(fmt,from,to)` - convert local to canonical format.
```
char *fmt,*from,*to;
```

The various message formats in this document are defined as binary C structures. Section 3.1.0 defines the canonical transmission format for these structures, and the routines *dkfcanon()* and *dktcanon()* help with conversions. For each routine the data at *from* is copied to the area at *to* under the interpretation of the null-terminated string *fmt*. Each routine returns the number of bytes moved to *to*. The format string is composed of the letters *c*, *h*, and *l*, which stand for byte, short, and long respectively. The copy proceeds by moving 1, 2, or 4 bytes for each *c*, *h*, or *l* encountered in *fmt* until the string is exhausted. By way of example a *dialout* structure (see section 3.1) could be converted from the local C represention to canonical represention as follows:

```
dktcanon("cchhhhhh",&msg,buf)
```

where *msg* is the name of the *dialout* structure and *buf* is the target buffer.

7. Acknowledgments

We are all indebted to A. G. Fraser for the inventions and philosophy behind Datakit and to him and J. R. Vollaro for bringing the hardware to life. I am particularly grateful to A. G. Fraser for support, encouragement, and technical discussions over an extended period of time. I am also indebted to R. J. Elliot, A. R. Koenig, P. M. Lu, R. Pike, D. M. Ritchie, P. J. Weinberger, and L. E. McMahon for contributions to software development and design.

G. L. Chesson

G. L. Chesson

8. Permuted Index

The following is a permuted index of terms plus the page numbers where they are discussed in sections 2 through 6.