DRAFT Design Specification for NI-CD ROM MARIAH Client Software V 1.0

31-MAR-1989

Written by:

Barbara C. Leahy—VMS Development

Issued by:

VMS DEVELOPMENT—ZK3-4

Review Team:

Elliott Drayton—VMS Development

Digital Equipment Corporation

CONTENTS

Preface

This is Design Specification describes the internal components that the development team will build for NI-CD ROM MARIAH Client Software V 1.0 This project conforms to DEC Standard 200 via the conformance of the controllers.

Associated Documents

1. Local Area Disk Protocol Architecture, Version 3.0, by Marshall Goldberg, 7-March-1989
2. Local Area System Transport Architecture, Revision 1.10, by: Bruce Eric Mann, Christian Saether, Philip Wells. 29-February-1988

Change History

Date	Issue #	Description/Summary of Changes
April 1989	Issue 1	Change from Rough Draft to Draft

1 System Introduction

This project supports software distribution from NI-CD ROM to the Mariah CPU. The product of this project will support initial software installation, layered product installation, and console patch updates from NI-CD ROM distribution media. The design of the product for this project has both CPU dependent areas and CPU independent areas. Therefore this project can be used on another CPU if the CPU dependent areas are appropriately substituted.

The goal of the NI-CD ROM project is to provide a simple Initial Software Installation process/interface for initial installation of software on the Mariah in a non-cluster environment. The Ethernet devices supported on the Mariah are the DEBNI and the XNA. Another goal of the MARIAH NI-CD ROM project is to supply a "simple-to-boot" user interface for booting a "Stand-alone backup" type of image. This simple-to-boot interface does not require the user to have to specify the controller board locations, XMI, and BI locations, nor the Ethernet Address of the host or target. Instead the user is only required to type in the service-name. A service-name is a name that specifies a particular compact disk on a server; service names are assigned at the time that a server is initialized. For details on the server, see the design specification document for the server.

It is not a goal of this project to support the booting of an operating system from NI-CD ROM.

1.1 Software Design Overview

The design for the project is a client/server relationship. The Mariah CPU is a client of a server box. The server box transfers blocks of data, from a compact disk, over the Ethernet wire to the client. The software on the remote server box is independent of the client CPU, and the data on the CD ROM is independent of the software running on the server. The design of the software for the client takes into account its independence from an operating system. The software must work for Ultrix, VMS, VAXELN, and the console patch code.

The elements of software on the NI-CD ROM project are:

- software on the block server
- software on the client CPU
- VMS operating system modules in the boot path
- console code

This design document only discusses the software on the client CPU. It does not go into a detailed explanation of the server software. See the design document on the server for details on that part of the project.

The specific modules for each component of the client are:

- Console port/class boot drivers
- VMB Transport class boot driver
- VMB Ethernet port boot driver
- VMB changes

- VMS booting path changes

1.2 System Interfaces

There are several types of interfaces for the client software. There are "user-interfaces" and "calling-interfaces" between the class and port boot drivers.
The interfaces are:

- the console level user-interfaces

  Show Services console command
  Find Services console command
  Booting command for booting from a server

- the coding interface in the console program

  Calling interface between console program and class driver.
  Calling interface between the class driver and the port driver.

- the architected port/class boot driver code not in the console

  Calling interface between VMB and the class driver
  Calling interface between the class driver and port driver

1.3 System Data Structures

The structures used by this code are the RPB (Restart Parameter Block), LAST message formats, and LAD message formats.

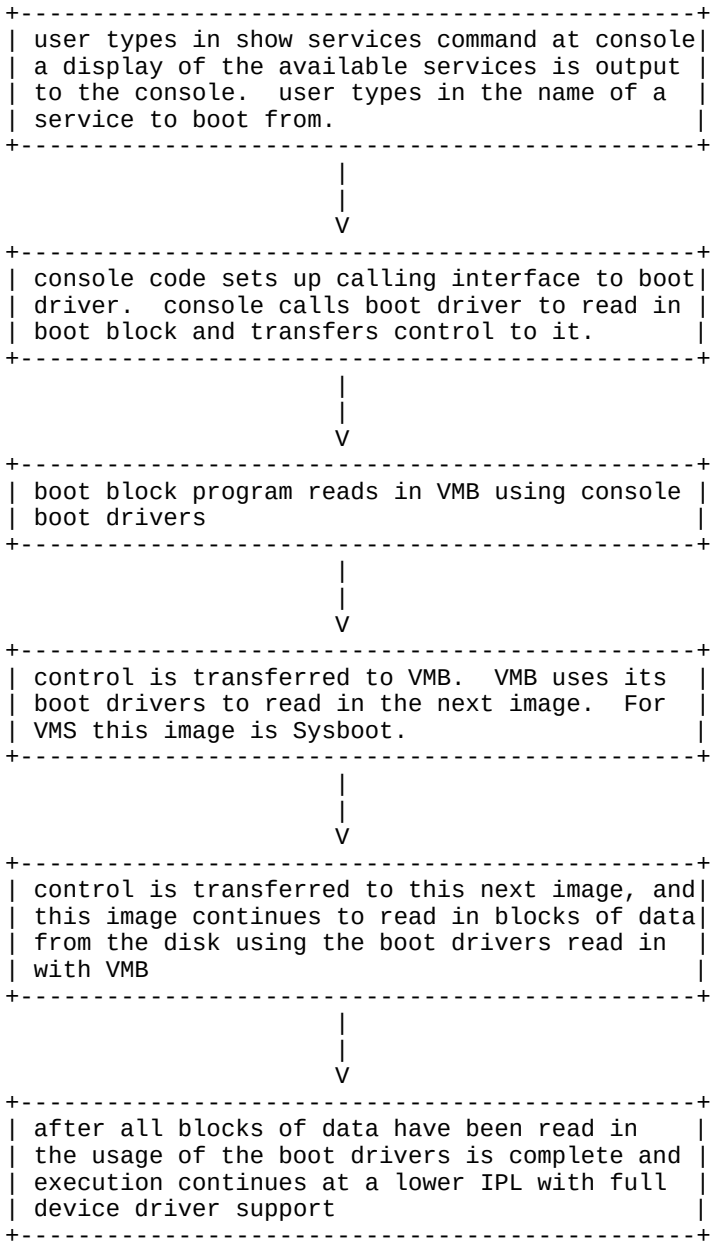
1.4 System Functional Design

The flow of control is typical of booting from a disk drive. The fact that the Ethernet wire is used is hidden from the user as much as possible.

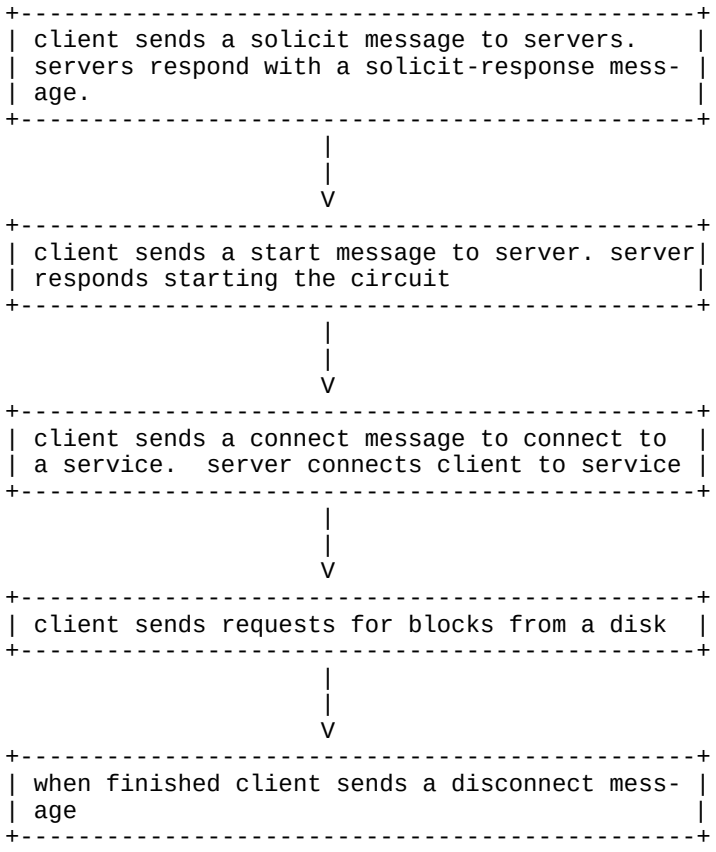
To boot, the user enters the name of the service after the boot prompt. The console program picks up the service name, finds the associated entry for the service name in its database, and obtains the necessary data for booting. The console then calls console port/class boot drivers to read in the boot block program. The boot block program uses the console boot drivers to read in VMB. VMB then takes over and uses its version of the port/class boot drivers to read in blocks from the CD ROM. For initial software installation, VMB is reading in a version of "Stand-alone backup" type of image that has NI-CD ROM support in it.

Once the client has NI-CD ROM a "Stand-alone backup" type of image on it, the initial software installation can continue. See the server documentation for details on installing software from NI-CD ROM.

A high level flow diagram of the booting process appears below. This diagram does not include the messages that go between the client and the server. It only shows an overview of the booting process.



Conceptually, the booting process for NI-CD ROM is the same process as booting from a disk. The difference is how that remote disk is accessed. The access to this remote disk occurs through the sending and receiving of commands and messages on the network wire. A master/slave or client/server relationship exists to supply this need. The client requests the servers to identify themselves with their Ethernet address. The client then sends a request to a particular server to start a circuit. After that circuit has been started, the client requests a connection to a service. Once the connection has been resolved, the client requests blocks of data from the service. After the client is finished receiving blocks of data, the client can end its association with the service by disconnecting the service and stopping the circuit. A flow diagram for these events, from the client's perspective, appears below:



1.5 System Design Criteria and Constraints

The software is designed to be fast and reliable. Booting time is always a consideration and hot issue. The code is designed to be faster than a TK50, when the TK50 is used as a console boot device.

The design is limited in areas of size, speed of the client processor, available memory, and IPL. Portions of the code are in console ROM and therefore are limited in size. The speed of the CPU is an issue. The booting path runs at IPL 31, so the drivers are built with that processing in consideration.

The calls from the port/class boot drivers are designed to ask for only 1024 bytes of data or 512 bytes of data. Requests for larger amounts of data would result in the boot drivers having to reorder the incoming requests since the protocol allows out-of-order returns. This feature of reordering of incoming requests is not supported by the class/port boot drivers.

1.6 System Test Specification

The system is tested by booting NI-CD ROM "Stand-alone backup" type of image and subsequently installing the software for the operating system from a disk. The client portion of the VMB booting software works when the NI-CD ROM "Stand-alone backup" type of image prompt appears. After the prompt the boot drivers are no longer used.

The console software is validated by observing the output from the "show services", "find services", and "boot service-name" commands. The features of the console command are validated by observing that all of the expected service-names appear on the console output. Expected service-names are names of the services provided by servers that have responded to the solicit message within the time limit. The console output must display these service-names only once per server. If two servers have duplicate service names, the console output must contain a duplicate indicator on the duplicate(s) service-name entry. Details on this are described further in the component sections.

If the system does not bring up NI-CD ROM "Stand-alone backup" there is a problem. Failure scenarios are:

- Server powering down
- Mariah CPU powering down
- Power glitches
- Bad media
- No communications path
- Others

1.7 System Environment

The physical environment consists of a Mariah CPU as the client and a system box containing a CD ROM drive connected to the Ethernet wire as the server.

1.8 Implementation Language

The VMB level software on the client is written in the Macro language. The console code is written in either BLISS32, Macro, or the language the console group has chosen.

1.9 Key Algorithms

Typical booting algorithms are used. See the LAST architecture for specific key algorithms of the transport architecture. See the LAD Architecture Specification for key algorithms on the Local Area Disk level.

1.10 Internationalization

The console group handles the messages. The server names are input by the user, they choose the language. Any naming restrictions or language restrictions already in evidence also apply to NI-CD ROM.

1.11 Costs

One engineer full time for one year for version one of the software. One assistant part time for one-half year for the internal and external field tests.

1.12 Design Constraints

The amount of ROM space on the Mariah CPU, and the speed of access from ROM, are issues for the boot drivers in the console code. Time to market is also an issue. The Mariah schedules require that the software be there in October of 1989.

2 Major Component Introduction

The major components of the entire CD ROM project are the server elements and the client elements. Only the client elements are discussed in this document. The client elements consist of the boot drivers, changes to VMB, and changes to the booting path in the VMS operating system.

As a reminder to the reader, all of these pieces of code run at IPL 31 and are therefore subject to the design limitations of that environment. In this environment interrupts are monitored by polling of registers, there is no WFIKPCH mechanism. Also, the coding considerations and design for the two components, software in ROM versus non-ROM software is different. These considerations might at first seem the same because of the IPL 31 constraint, but the console code has other constraints as well. The console code must be small, it must be either fast, or be copied into some memory that has been enabled. The code must be simple in its paths so that all paths can be tested. It must be reliable, predicatable and bug free to the best possible extent. This is a requirement because of the cost of updating ROMs, and the number of ROMs.

2.1 Component Software Design Overview

The "booting" client software consists of the software necessary for installing software from a CD ROM disk over the Ethernet wire. This includes port/class boot drivers in the console code, port/class boot drivers read off of the CD ROM, and the operating system routines.

The software design used for the boot drivers is a port/class architecture. A class driver creates messages, gives those requests to the port driver to send over the wire, commands the port driver to listen for packets on the wire, and commands the port driver to initialize the port. The port driver is responsible for initializing the port and handling all of the I/O to and from the port. The port driver has no context of the messages being sent to and from it by the class driver.

The booting client software is running in a boot environment and as such runs under the following environment and constraints:

- IPL 31
- Single threaded, deterministic in nature
- Both physical mode and virtual mode addressing
- Runs without an operating system
- Reads in the operating system or designated program

2.2 Component Interfaces

The port/class boot drivers, above the console level, interface with both operating system independent routines and operating system dependent routines. The operating system independent routines are VMB and the routines that make up the VMB executable. Some of the VMS operating system dependent routines are Sysboot, Sysinit, parts of Stand-alone backup, and other operating systems routines.

2.2.1 Console Calling Interface

There are many calling interfaces at the console level. There are interfaces between:

- Console and Class boot driver
- Boot Block Program and Class Driver
- Console and Boot Block Program
- Class boot driver and Port boot driver

For each interface listed above a description of the calling parameters is described in this section.

The console calls the class boot driver to perform the operations:

- FIND SERVICES
- SHOW SERVICES
- BOOT Service-Name

Other commands that are optionally are:

- SHOW SERVER
- SHOW SERVER/DEVICE
- SHOW SERVER/DEVICE/AVAILABLE
- ALLOCATE servername/DEVICE=device#
- DEALLOCATE servername/DEVICE=device#

The class boot driver consists of the routines:

- Initialization
- Disconnect Routine
- Main Action Routine

  Entry point for Console program
  Entry point for Boot Block program

The input the console supplies to the class driver's initialization routine is:

```
;+
; CLASS DRIVER INITIALIZATION ROUTINE
;
; Inputs:
;
; R2: CSR
; R3: STARTING LOCATION OF ETHERNET PORT DRIVER
; R4: TABLE IN ETHERNET PORT DRIVER CONTAINING OFFSETS
;     TO THE ROUTINES IN THE DRIVER
;
; Outputs:
;
; R0: SS$_NORMAL
;     SS$_NOSUCHDEV
;     SS$_ABORT
;     SS$_CTRLERR
;     SS$_CONNECFAIL
;
;
;--
```

The input the console supplies to the class driver's disconnect routine its:

```
;+
;       CLASS DRIVER DISCONNECT ROUTINE
;
; Inputs:
;
; R2: CSR
;
; Outputs:
; R0: (status)
;     SS$_NORMAL
;     SS$_ABORT
;
;
;--
```

The Main Action Routine must have two entry points, one for the the console program and the other for the boot block program. The class boot driver reads only one block at a time, and the size of the block is 512 bytes. When the console program calls the class boot driver to read in the boot block program, it supplies the class boot driver with the input:

```
;++
; Console Class Driver Main Routine
;
;
;
;*** CONSOLE CALLING PROGRAM ENTRY REGISTER SETTINGS: ***
;
; - For SHOW SERVICES -
; - For FIND SERVICES -
; - For SHOW SERVERS -
; - For SHOW SERVERS/DEVICE -
; - For SHOW SERVERS/DEVICE/AVAILABLE -
;
```

```
; R0:  Boot Device Type - BTD$_SERVER_DEBNI
;                         BTD$_SERVER_DEBNA
;                         BTD$_SERVER_XNA
;
; R2:   CSR
;
; R3:   Base of good memory to deposit blocks Available Target
;       Identifiers (ATI).
;
; R4:   QIO qualifier  - IO$_ACCESS
;                       IO$_CREATE
;                       IO$_NETCONTROL
;
; R6:   Size of data base
;
; R7:   Time to wait for first response from find services command
;
; R8:   Maximum time to wait for all responses from find service
;       command
;
; - For Reading in the Boot Block Program
;        When the console program calls the class boot to read in
;        the boot block program it must supply the following register
;        values in addition to those shown in the general section above.
;
;
; R0:   Boot Device Type - BTD$_SERVER_DEBNI
;                          BTD$_SERVER_DEBNA
;                          BTD$_SERVER_XNA
;  R2:   CSR
;
;  R3: Address of Available Target Identifier that contains:
; Source (Client) Hardware Ethernet Address
; Destination (Server) Hardware Ethernet Address
;
; R5:   Buffer address of where to write boot block program
;
; R7: Block size, default value is 512
; R8:   Logical block number
;
;
```

The console program must also set up the calling interface for the boot block program. When the console is going to do a boot from a service it calls the the boot block program with the information below. The boot block program uses this information to find the class boot driver and issue requests to it.

CALLING INTERFACE BETWEEN THE CONSOLE PROGRAM

AND

THE BOOT BLOCK PROGRAM

```
;++
; Parameters supplied to the boot block program so that it
; can locate the class boot driver and use it to read in
; VMB.  Note that all reads performed are implicitly 512 bytes
; in length.
;
; R0:   Boot Device Type - BTD$_SERVER_DEBNI
;                          BTD$_SERVER_DEBNA
;                          BTD$_SERVER_XNA
; R2:   CSR
;
; R3: Address of Available Target Identifier that contains:
; Source (Client) Hardware Ethernet Address
; Destination (Server) Hardware Ethernet Address
;
; R5:   Software boot control flags
;
; R6:   Physical address of the server class boot driver
;       that reads an arbitrary Logical Block Number into
;       memory.
;
; SP:   <BASE_ADDRESS + ^X200> of 64 kb of good memory
;
```

The boot block programs calls the console class boot driver to read in VMB. The boot block program calls into the main routine of the class boot driver, but it uses a different entry point than the console program. The calling interface between the Boot Block program and the class boot driver is shown below.

```
;++
; Calling interface between Boot Block Program and Console
; Class boot driver
;
; Inputs:
;          R2 - physical address of boot device's CSR
;          R3 - Available Target Identifier for boot
;          R5 - starting address of transfer(relative to load
;               address)
;          R8 - LBN to transfer from boot device
;          4(SP) physical address of transfer
;
; R7-R9 are scratch registers
; R0 is status, SS$_NORMAL (1) is success, Low bit is cleared on
;    failure
; R1-R6,R10-R11,AP, and SP are preserved
;
```

The class driver sets up any necessary registers and fields for the Ethernet port driver. The information passed to the port driver is:

- Packet to be sent
- Flag indicating request is a Remote Access Protocol (RAP) request and not a MOP Protocol request (if necessary)
- Software boot control flags
- Base of good memory

2.2.2 VMB and higher level Calling Interfaces

Above the console calling interface are VMB and operating system routines. At this level there is: the calling interface between a program and the class boot driver, and the calling interface between the class and port boot drivers.

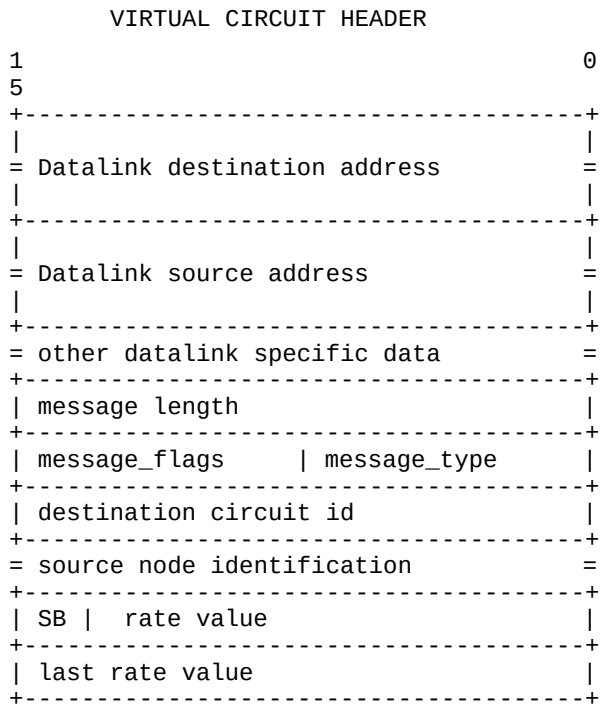
The class driver has an initialization routine, main routine, and a disconnect routine. The BOO$QIO interface is the calling interface to the main routine of the boot driver. This "main" routine performs the sending and receiving of blocks of data to/from the server. The calling interface to the initialization routine is the standard calling interface along with information needed for accessing the Ethernet driver. The information passed to the intialization routine is:

- Port information, XMI,BI
- RPB
- Information in the RPB about the server to boot from, Ethernet address, name, disk number, boot device type
- Location of the port boot driver

2.3 Component Data Structures

The class boot driver uses the LAST/LAD protocols to transmit messages to the server. Only a limited subset of LAST messages are used by the class drivers. The format of the messages used are shown by the structures presented in this section. For each structure a block diagram and macro template is presented. The macro template is only an outline. These structures are either from the LAST Architecture Specification or the LAD Protocol Specification.

All LAST message are preceeded by a virtual circuit header. The format of this header is shown below.



The format of the fields, per the LAST architecture specification, are:

```
Message length:          2 bytes unsigned
                         length of the message to follow
Message type:            1 byte
                         0 = run message
                         1 = start message
                         2 = stack message
                         3 = stop message
                         4 = loop message
                         5 = advertisement
                         6 = solicit message
                         7 = solicit response message
Message flags:           1 byte, bit 0 = 1, checksumming is done
                         bits 1-7, must be zero
Destination Circuit Id:
                         2 bytes
                         reference to destination node connection
                         block structure
Source Node Identification:
                         6 bytes
                         Unique identification of the
                         source node which remains constant across
                         transport incarnations.
SB:  1 bit               If set indicates the rate value is being established.
Rate Value:              15 bits, segment per second
Last Rate Value:         16 bits unsigned, segment per second
```

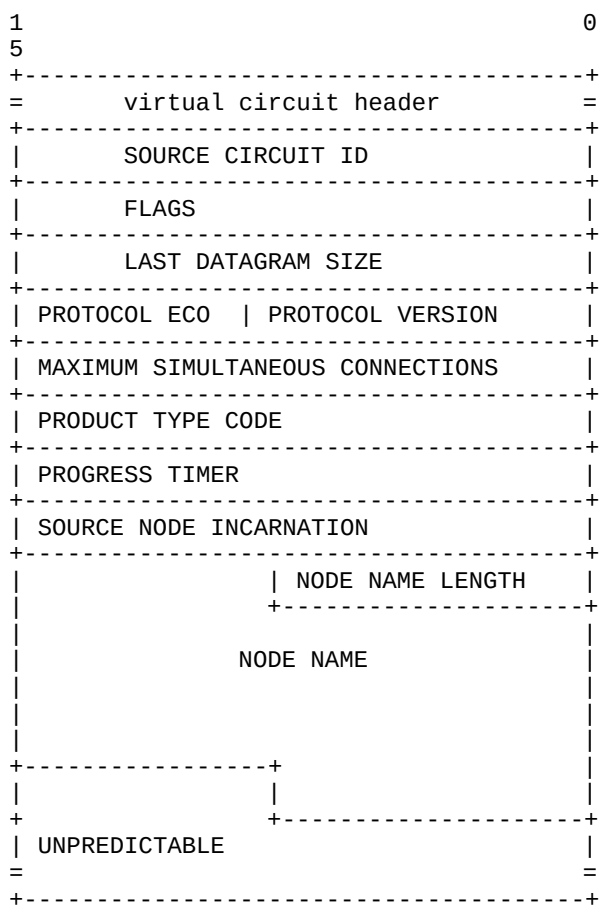
A basic macro layout for the header format is shown below. This header precedes the message formats for any of the messages sent by the client.

```
;++
; basic template for virtual circuit headers
;
;--
MESSAGE_LENGTH: .BLKW 1  ; unsigned length of message to follow
MSG_TYPE:       .BLKB 1  ; message type
                         ;       0 = run
                         ;       1 = start
                         ;       2 = stack
                         ;       3 = stop
                         ;       4 = loop
                         ;       5 = advertisement
                         ;       6 = solicit (YES)
                         ;       7 = solicit response (YES)
MSG_FLAGS:      .BLKB 1  ; bit 0 set indicates data beginning
                         ; at message_length through message_length
                         ; bytes will be checksummed
                         ; bits 1-7 MBZ
DESTINATION_CIRCUIT_ID:
                .BLKW 1  ; reference to destination node
SOURCE_NODE_IDENTIFICATION:

                .BLKB 6  ; Unique ID of source node which remains
                         ; constant across transport incarnations
                         ; The Ethernet address of the client is
                         ; put into this field
RATE_VALUE:     .BLKB 1  ; Bits 0-14, a segment per second count
                         ; Bit 15, when set indicates rate value is
                         ; being established
LAST_RATE_VALUE:.BLKW 1  ; Segment per second count
```

Following this header are the LAST messages. These messages are used by the boot drivers to "start a transport","connect to a service","receive data", and "solicit servers". The formats of these messages are shown below:

The start message format is:

START MESSAGE



The macro template for the Start message is:

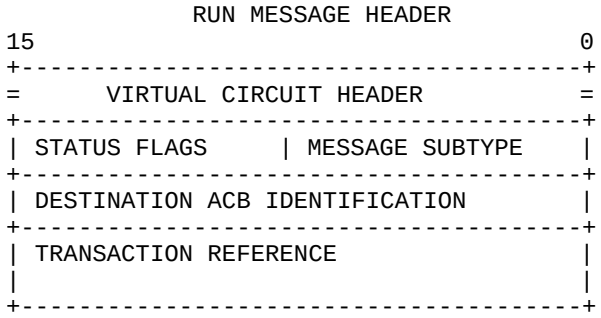
```
;+
; Template for the LAST start/stack message
;
; This message must be preceeded by the virtual circuit header
; outlined above.  Name are puposedly spelt out here to avoid
; confusion, they can be made smaller for usage in the code by
; any coder using the templates.
;
;-

SOURCE_CIRCUIT_ID:      .BLKW 1 ; "Reference to source node CSB
                                ; structure"
FLAGS:                  .BLKW 1 ; "Flags used to negotiate circuit level
                                ; features"
                                ; "Bit 0 - intra burst delay"
                                ; "Bit 1 - force checksumming "
                                ; "Bits 2-15 MBZ"
LAST_DATAGRAM_SIZE:     .BLKW 1 ; Size of frames
PROTOCOL_VERSION:       .BLKB 1 ; number indicating protocol used
PROTOCOL_ECO:           .BLKB 1 ; patch updates to software
MAXIMUM_SIMULTANEOUS_CONNECTS:  ; unsigned value of the maximum
                        .BLKW 1 ; number of associations that can
                                ; be opened on this virtual circuit
PRODUCT_TYPE_CODE:      .BLKW 1 ; unsigned value designating the product
PROGRESS_TIMER:         .BLKW 1 ; unsigned value specified in seconds
SOURCE_NODE_INCARNATION:.BLKW 1 ; "unique value assigned at transport
                                ; initialization"
NODE_NAME_LENGTH:       .BLKB 1 ; length of the node name
NODE_NAME:              .BLKB 16; Last supports name up to sixteen
                                ; characters in length.  Note that
                                ; the server name is limited to
                                ; "6" characters for version one.
```

This node name length is under review for revision to be compatible with decnet node names of length 255.

The specific values used for this project are:

```
SOURCE_CIRCUIT_ID:      .WORD ^X0BCD
FLAGS:                  .WORD 1 ; wait for status
LAST_DATAGRAM_SIZE:     .WORD 1500
                                ; only want to transfer 1024 bytes
                                ; per packet
PROTOCOL_VERSION:       .BYTE 3 ; current level
PROTOCOL_ECO:           .BYTE 0 ; current level
MAXIMUM_SIMULTANEOUS_CONNECTS:
                        .WORD 2 ; allow for console level and VMB level
                                ; if protocol broken, then must set
                                ; to infinity
PRODUCT_TYPE_CODE:      .WORD 10; booting client
PROGRESS_TIMER:         .WORD 60; value in seconds
SOURCE_NODE_INCARNATION:.WORD 0 ; value assigned at transport
                                ; initialization
```

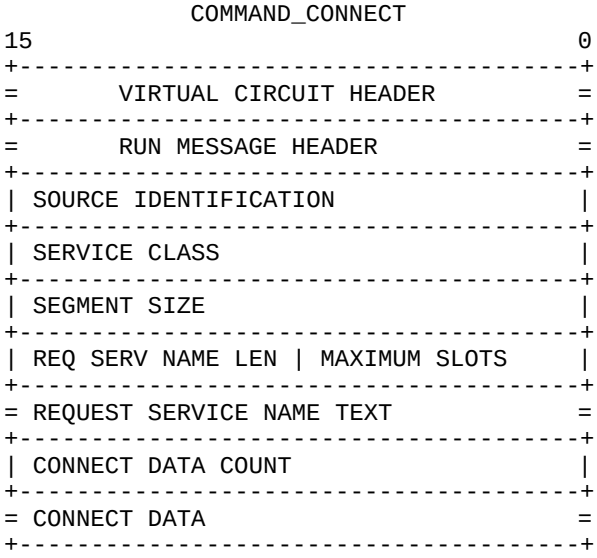
The start message is associated with setting up the transport circuit sub-layer (See LAST Architecture Specification for a description of the circuit sub-layer). This message can now be followed with messages from the transport association sub-layer. Messages at this level perform the functions of: connect to the service, disconnect from the service, request data from the service, or resync with the service. These messages are "run" messages and are preceeded by a "run message" header. The format of the "run message" header is:



The template for the run message header appears below.

```
;+
;Run Message Template Header
;-
MESSAGE_SUBTYPE:      .BLKB 1   ; type of message that follows
                                ; 0 = COMMAND_DATA
                                ; 1 = RESPONSE_DATA
                                ; 2 = COMMAND_CONNECT
                                ; 3 = RESPONSE_CONNECT
                                ; 4 = COMMAND_RESYNC
                                ; 5 = RESPONSE_RESYNC
                                ; 6 = COMMAND_DISCONNECT
                                ; 7 = RESPONSE_DISCONNECT
STATUS_FLAGS:         .BLKB 1   ; modifiers
                                ; bits 0-1, mode indicator
                                ;       mode 0 = idempotent
                                ;            1 = timed
                                ;            2 = normal
                                ;            3 = datagram
                                ; bits 3-7 must be zero
DESTINATION_ACB_IDENTIFICATION:
                        .BLKW 1   ; Index to server structure.
                                ; "Must be zero for a connect
                                ; request message, and must not
                                ; be zero for any other run
                                ; message"
TRANSACTION_REFERENCE:.BLKL 1  ; A client specified value.
```

The run message sub-types used by the boot drivers are: connect, and data request. The format for the command_connect message is shown below.



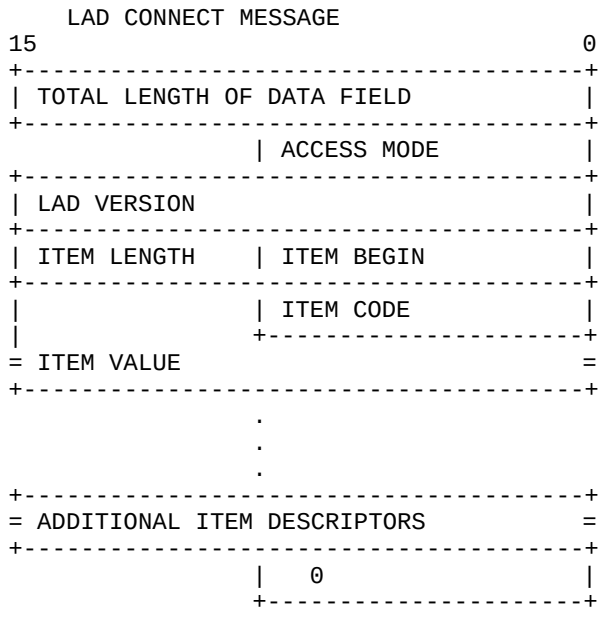
The macro template for the command_connect message is below.

```
;+
; Command connect template header
;
;-
SOURCE_IDENTIFICATION:
                        .BLKW 1 ; unsigned value
SEGMENT_SIZE:           .BLKW 1 ; segment size used to fragment
                                ; transaction requests into.
MAXIMUM_SLOTS:          .BLKB 1 ; Maximum number of transactions
                                ; that can be pipelined by a client
REQUEST_SERVICE_NAME_LENGTH:
                        .BLKB 1 ; count of request service name
REQUEST_SERVICE_NAME_TEXT:
                        .BLKB 6 ; service name
DATA_COUNT:             .BLKB 2 ; byte count of application data
CONNECT_DATA:                   ; data to follow
```

The default values for this project are:

```
SOURCE_IDENTIFICATION:  .WORD ^X0BCD
SEGMENT_SIZE:           .WORD 1460 ; segment size used to fragment
                                ; transaction requests into.
MAXIMUM_SLOTS:          .BYTE 1 ; Maximum number of transactions
                                ; that can be pipelined by a client
```

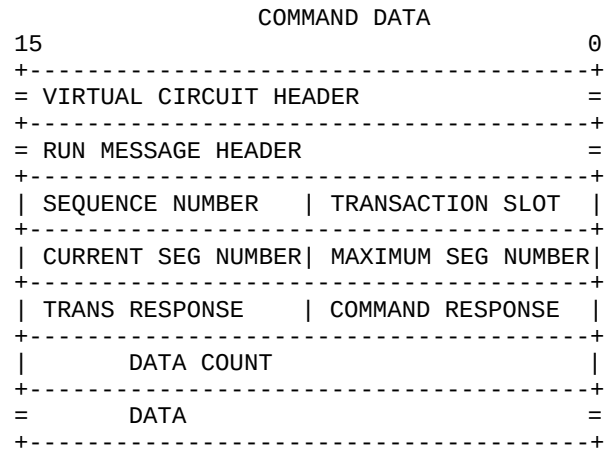
The LAST connect message contains a LAD protocol message within its data area. The LAD protocol message is used to establish an association with a LAD service. The format of this message is below:



The macro template for this header is below:

```
LAD_LENGTH:       .BLKW 1  ; total length of data field
LAD_ACCESS_MODE:  .BLKB 1  ; mode of access
                           ; 0 = write access is required
                           ; 1 write access is not required
                           ; 2 write access is requested, read access
                           ; is acceptable
LAD_VERSION:      .BLKW 1  ; version of LAD supported
ITEM_BEGIN:       .BLKB 1  ; LAD item list type, = 2 for this message
ITEM_LENGTH:      .BLKB 1  ; Length of item descriptor
ITEM_VALUE:       .BLKB 39 ; This is the maximum size
ITEM_CODE:        .BLKB 1  ; type of item being referenced
```

The other LAST run message used by the client is the command_data message. The format of the command_data message is shown below.

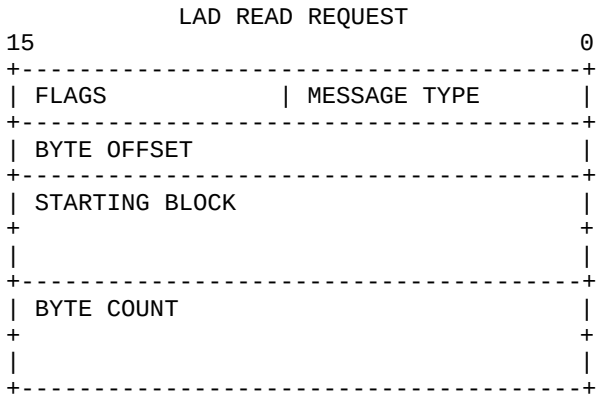


A macro template for the command_data format appears below.

```
;+
; template for command_data message
;-
TRANSACTION_SLOT:       .BLKB 1 ; current transaction slot
SEQUENCE_NUMBER:        .BLKB 1 ; number for this sequence
MAXIMUM_SEQUENCE_NUMBER:.BLKB 1 ; unsigned, number of segments in command
CURRENT_SEGMENT_NUMBER: .BLKB 1 ; unsigned, segment number
COMMAND_RESPONSE:       .BLKB 1 ; unsigned value of command response timer
TRANSACTIOPN_RESPONSE:  .BLKB 1 ; unsigned transaction response timer
DATA_COUNT:             .BLKW 1 ; unsigned value of the bytes of data
DATA:                           ; user supplied data
```

In addition to the LAST transport messages are the LAD protocol messages. LAST messages control the transport layer. LAD protocol messages are the actual messages that transmit the requests and responses for data to/from the disk device. The LAD protocol messages occupy the first bytes of the data sections in the LAST messages.

The client requests the server to read a block of data from the disk by sending a LAD read /write request. The format of this request is:

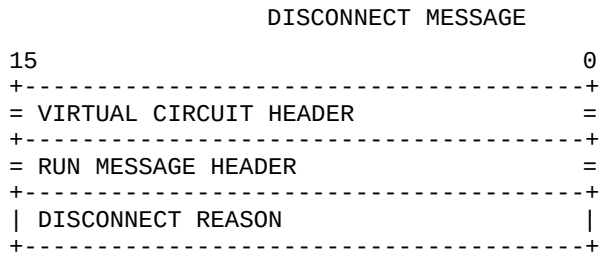


A macro template for this packet of information follows.

```
;+
; LAD protocol read request header
;-
MESSAGE_TYPE:           .BLKB 1 ; value of 2 indicates a read request
FLAGS:                  .BLKB 1 ; MBZ for read requests
BYTE_OFFSET:            .BLKW 1 ; must be zero for our application
STARTING_BLOCK:         .BLKL 1 ; starting block number to read
BYTE_COUNT:             .BLKL 1 ; number of bytes to transfer
```

This header is located in the data area of the LAST command_data message.

After the block transfers are over, the client sends a command disconnect message to end a session. The format of that command is:

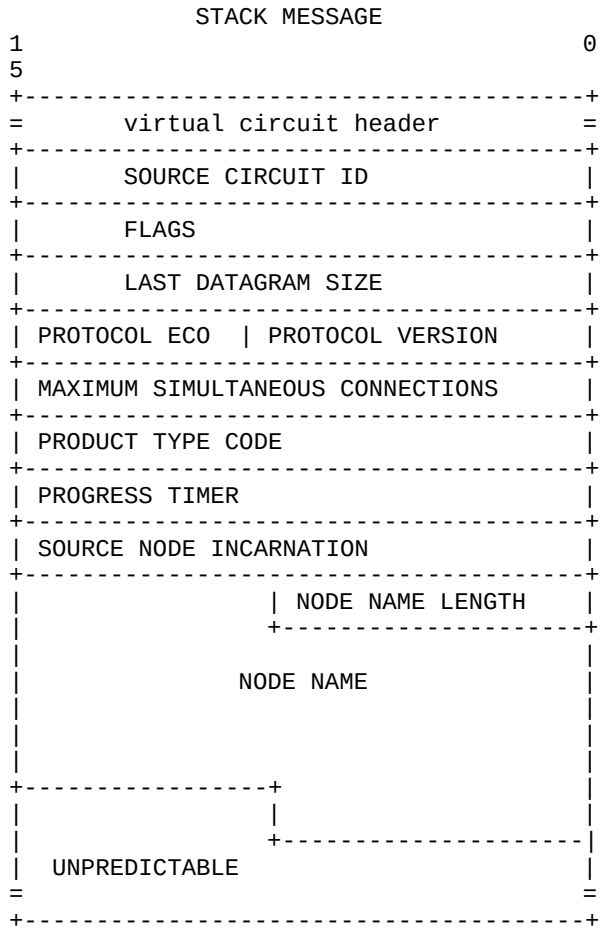


The macro template for the command is below:

```
;+
; DISCONNECT COMMAND
;-
DISCONNECT_REASON:          .BLKW 1 ; unsigned value, disconnect reason
```

The messages discussed to this point have all been sent by the client. This next section displays pertinent structures "received" by the client. These mesages are the "stack" message, "data response" message and the "stop" message. The formats of these messages are listed below.

The stack message has the same format as the start message, except that the message type in the VC header is set to the value of two. The format is:



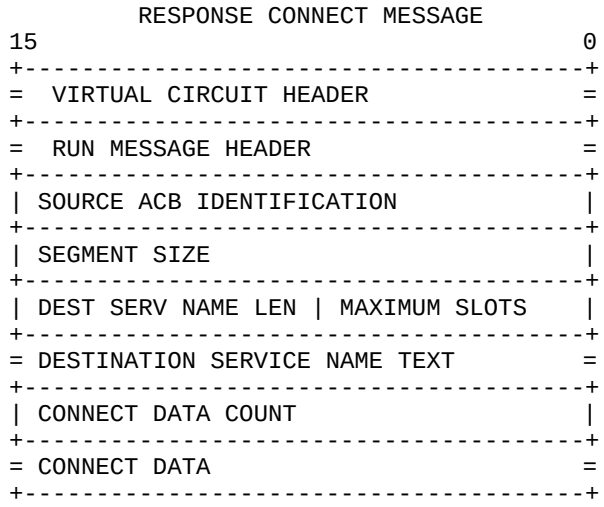
The macro template for this message is below.

```
;+ Template for the LAST start/stack message
;
; This message must be preceeded by the virtual circuit header
; outlined above.  Names are purposely spelt out here to avoid
; confusion, they can be made smaller for usage in the code by
; any coder using the templates.
;
;-
SOURCE_CIRCUIT_ID:      .BLKW 1 ; "Reference to source node
FLAGS:                  .BLKW 1 ; "Flags used to negotiate circuit level
                                ; features"
                                ; "Bit 0 - intra burst delay"
                                ; "Bit 1 - force checksumming "
                                ; "Bits 2-15 MBZ"
LAST_DATAGRAM_SIZE:     .BLKW 1 ; Size of frames
PROTOCOL_VERSION:       .BLKB 1 ; number indicating protocol used
PROTOCOL_ECO:           .BLKB 1 ; patch updates to software
MAXIMUM_SIMULTANEOUS_CONNECTS:  ; unsigned value of the maximum
                        .BLKW 1 ; number of associations that can
                                ; be opened on this virtual circuit
PRODUCT_TYPE_CODE:      .BLKW 1 ; unsigned value designating the product
PROGRESS_TIMER:         .BLKW 1 ; unsigned value specified in seconds
SOURCE_NODE_INCARNATION:.BLKW 1 ; "unique value assigned at transport
                                ; initialization"
NODE_NAME_LENGTH:       .BLKB 1 ; length of the node name
NODE_NAME:              .BLKB 16; LAST supports names up to sixteen
                                ; characters in length.  Note that
                                ; the first version of the server name
                                ; is limited to less than this value.
```

The node name field size is under review for upgrade for compatibilty with DECNET Phase V.

The client receives LAST run messages with the subtypes: response_connect, data_response, resync response, and stop.

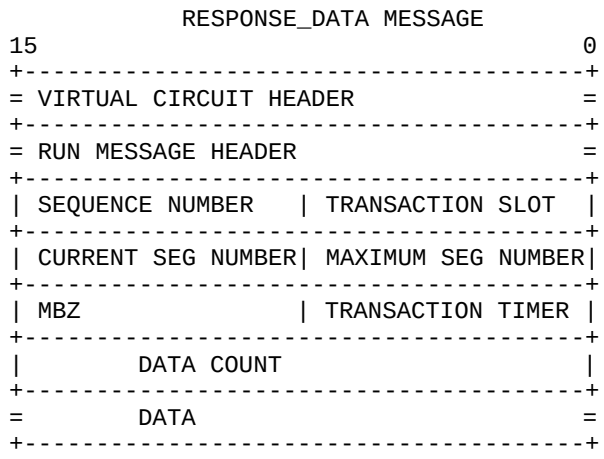
The format of the response_connect message is:



The macro template for the response_connect message is below:

```
;+
; Response Connect template header
;
;-
SOURCE_ACB_IDENTIFICATION:
                        .BLKW 1 ; unsigned index to connect block
SEGMENT_SIZE:           .BLKW 1 ; segment size used to fragment
                                ; transaction requests into.
DESTINATION_SERVICE_NAME_LENGTH:
                        .BLKB 1 ; count of request service name
DESTINATION_SERVICE_NAME_TEXT:
                        .BLKB x ; service name, x = size
DATA_COUNT:             .BLKB 2 ; byte count of application data
DATA:                   ; data to follow
```

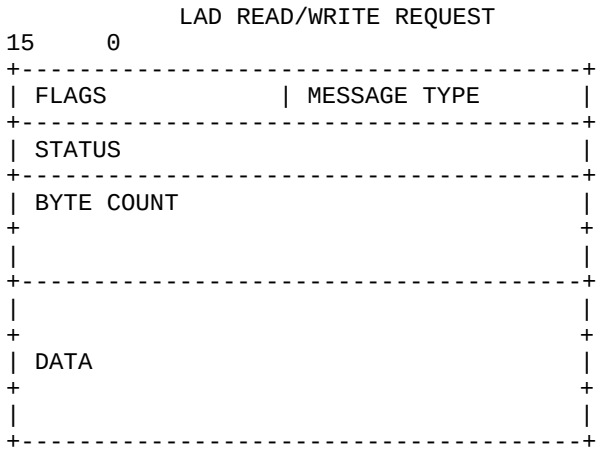
The client receives blocks of data in a response_data message. The format of the response_ data message is:



A macro template for the response_data command format appears below.

```
;+
; template for response_data command
;-
TRANSACTION_SLOT:       .BLKB 1 ; current transaction slot
SEQUENCE_NUMBER:        .BLKB 1 ; number for this sequence
MAXIMUM_SEQUENCE_NUMBER:.BLKB 1 ; unsigned, number of segments in command
CURRENT_SEGMENT_NUMBER: .BLKB 1 ; unsigned, segment number
TRANSACTION_TIMER:      .BLKB 1 ; unsigned value of command response timer
MUST_BE_ZERO            .BYTE 0 ; unsigned transaction response timer
DATA_COUNT:             .BLKW 1 ; unsigned value of the bytes of data
DATA:                           ; user supplied data
```

LAD protocols are also in the data areas in the LAST data_response message. The format of the LAD protocol for requesting a disk read or write is shown below. The NI-CD ROM project only supports a read request. It is an error for the server to receive a write request from the client and it is also an error for the the client to receive a write response. For a read request the message type field in the LAD message has the value of 4. A write response has the value of 5.

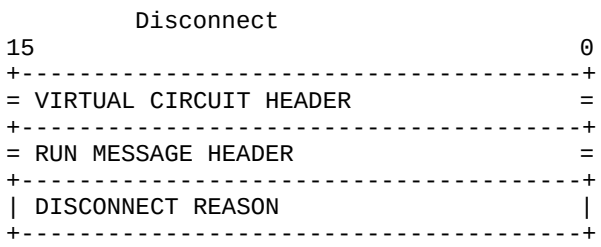


A template for this packet of information follows.

```
;+
; LAD protocol Read Request Header
;-
MESSAGE_TYPE:           .BLKB 1 ; value of 2 indicates a read request
FLAGS:                  .BLKB 1 ; MBZ for read requests
STATUS:                 .BLKW 1 ;
BYTE_COUNT:             .BLKL 1 ; number of bytes to transfer
DATA:
```

A resync response message can be recieved by the client. This message is not handled at this time.

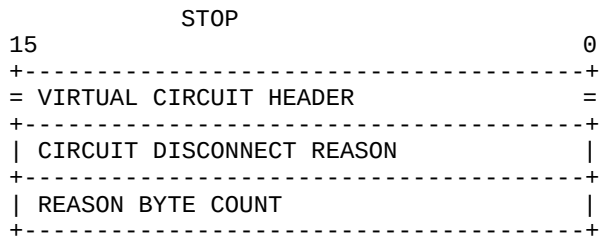
The format of the response disconnect message is:



The macro template for this command is listed below:

```
;+
; RESPONSE DISCONNECT MESSAGE
;-
DISCONNECT_REASON:      .BLKW 1 ; Reason for disconnect
```

The client can also receive a circuit sub-layer stop message. The format for this message is:



The macro template for the stop message appears below:

```
;+
; Stop Message format
;-
CIRCUIT_DISCONNECT_REASON:      .BLKW 1 ; unsigned value
REASON_BYTE_COUNT:              .BLKW 1 ; no value defined
```

2.4 Component Functional Design

Two components of the NI-CD ROM project are discussed in this section. Those components are the VMB port/class boot drivers and the console port/class boot drivers. The other component of the NI-CD ROM project, the server, is discussed in the "Server Design Specification". Please refer to that document for details on the design.

The console port/class boot drivers reside on ROM and are responsible for reading in the Boot Block program and VMB. The extent of this code is limited. The console class driver implements the LAD and LAST protocols necessary to operate under this enviromnent. This driver, called the ACBTDRIVR, only performs the operations necessary to query the network to find the services available, connect to a particular service, and read blocks from that service.

After VMB has been read in, the console transfers control to it. VMB issues BOO$QIO calls to the class boot driver to read in the next layer of software in the boot process. The port/class boot drivers that VMB uses for NI-CD ROM boot are the BEBTDRIVR and the associated Ethernet port driver. BEBTDRIVR handles issuing the requests to and from the server. It issues messages to start a transport, connect to a service and read blocks of data from the service. All of the messages sent to and from the BEBTDRIVR are first given to the Ethernet port driver for dispatching or receiving.

After VMB has read in the next module in the booting path it transfers control to it. This module is Sysboot for VMS. Successive booting modules call BEBTDRIVR to read in more blocks of data. This continues on until an operating system or an operating system level program ("Stand-alone backup" type of image) takes over. (This typically occurs when IPL is dropped below 31.) At that time run-time versions of the LAST/LAD drivers are used.

Since there is such a distinct difference in the port/class boot drivers used at the console level versus the VMB level, the discussion on them has been broken up into two sections, the console drivers, and the VMB port/class boot drivers.

2.4.1 CONSOLE Port/Class Boot Drivers

The console port/class boot drivers perform two basic functions. They perform the actions necessary to obtain a display of the available services for booting, and they read in the boot block and VMB. These two operations have been separately discussed in two sub-sections below. One section is "console commands" the other is "booting".

2.4.1.1 Console Commands

Commands exist in the console to display to the user information about the services for booting and the server. These command are:

- "FIND SERVICES"
- "SHOW SERVICES"

Optional commands at the console are:

- "SHOW SERVERS"
- "SHOW SERVERS/DEVICES"
- "SHOW SERVERS/DEVICES/AVAILABLE"
- "ALLOCATE servername/DEVICE=device_number_identifier"

- "DEALLOCATE servername/DEVICE=device_number_identifier"

Each of these commands outputs information to a user at the console terminal or provides a service to the user. The "FIND SERVICES" command provides information to a user about available services. It queries all the servers on its network for information about services available. A service is defined as a compact disk in a compact disk drive on the server. This disk is spinning and available for access by an appropriate client. The "find services" command is similiar to the "show services" command. It also displays the available service-names for booting. The difference is that the "find services" command queries the network for the available services, and loads this information into a database within the client; whereas the "show services" command does not query the network and instead only outputs entries in the clients database. Both commands share the same printout format. The "show services" command is faster since it does not involve any network activity; however, the entries in the database may be out of date and not reflect the network. Entries created by the "find services" command reflect the service-names on the network at the time of the command. Therefore the user has the choice of examing the network for the available services, or seeing what services the machine knows about already and thereby not incurring the time penalty of searching the network. These names are printed on the console terminal in alphabetical order.

The "show services" command provides information to a user about known available services that the console currently knows about. The console displays the services in its local database to the screen, it does not query the network asking for services. Its goal is to add some user friendliness to booting. This command is similiar to a "show devices" command for devices attached to a system. The output of the "show service" command is a display of the "available service-names for booting a Stand-alone backup type of image" on the console device.

The other commands, "show servers, show servers/devices, show servers/ devices/available, allocate servername/device=#", all interact with the server, and not a service on the server. These commands are used to find information about the server, what devices are on the server, and the state of those devices. The allocate commands also allows the user to reserve a driver for his/her use. The "show servers" commands queries the network for servers on the network, and receives messages back from every server that answers the enquery. The console program then formats the output and displays it onto the screen. The format of this display for two server entries is shown below. Note that entries are separated by a blank line.

```
>>> SHOW SERVERS

    SERVER NAME
    ETHERNET ADDRESS
    Server description string

    SERVER NAME
    ETHERNET ADDRESS
    Server description string
```

The "show servers/devices" command lists the devices attached to each server answering the inqury. The state of the device is listed along with the information on that device. A device has characteristics defined for it:

- unavailable (dead hardware)
- available (no disk inside, assumed good hardware state)

- allocated (drive is allocated,no disk inside)
- loaded (disk is spinning, no connections associated with it)
- active (disk has outstanding connections to it)

A device is UNAVAILABLE if it does not respond to a command by the operating system, or responds with a message indicating it is in a bad state. A device is AVAILABLE if it has no disk inside of it, is in good hardware state implying that it responds to operating system commands, and is not allocated. A device is ALLOCATED when a user issues an allocate command specifying the name of the server that device is on, and the unit number identifier of that device. Note that a user can issue an allocate command to the drive only when drive is in the available state and at no other time. A device is LOADED if a compact disk is inside of it that is spinning, and there are no connections. A connection is defined as a user gaining access to a service provided by the disk. If a disk is not allocated, and there are no connections on it, then it is not being used as a service and can be taken out of the drive without disrupting any service activity. A device is ACTIVE if it has a disk spinning inside of it which is being accessed, therefore has connections and is thus providing a server to a user.

All of the "show server" commands and options give the user information about the state of the server and devices on that server. To boot from a server a device must be in either the loaded or active states, implying there is a compact disk in the driver. It is not necessary for a user to allocate a drive on a server prior to using it. Instead the allocate command is provided as an option for the user to "reserve" a drive for his/her own compact disk.

The format for the "show servers/device" and show servers/device/available" commands are shown below:

```
>>> SHOW SERVERS/device

    SERVER NAME
    ETHERNET ADDRESS
    Server description string
    DEVICE NAME    ASSIGNED DEVICE NUMBER      DEVICE CHARACTERISTICS
```

The "assigned device number" can be different from the actual device number of the disk. By default the assigned device number for a disk is the number associated with the disk, unless overriden by a management program. For example, DKA300 would have a default "assigned number" of 300 as shown in the diagram below. Also in the diagram below is an example of a disk that uses an assigned value instead of the default value; DKA700 has an assigned number of 1, instead of the default of 700.

```
>>> SHOW SERVERS/DEVICE

 TONY
 80-00-00-00-00-00
 DEVICE: Name   Assigned Number     Characteristics
     DKA300              300     ALLOCATED
            DKA700              1              AVAILABLE
```

The "show servers/device/available" has the same display format as the "show servers/device" command with the difference being that only available devices are shown. For example, the output for a "show servers/device/available" command for the data above is:

```
>>> SHOW SERVERS/DEVICE/AVAILABLE

 TONY
 80-00-00-00-00-00
 DEVICE: Name   Assigned Number
            DKA700              1
```

Notice in the example that device DKA300 does not appear in the display since it is not available for use.

A user can "allocate" or "deallocate" a device. This allows the user to reserve and unreserve a device for his/her use. It is useful when the server is a long distance, possible on another floor from the console and the user wants to ensure that the drive will be his/hers when he arrives at the server.

To boot from a server the user types the "find service" command with pertinent qualifiers at the console prompt to see information about the servers. Then, to boot from NI-CD ROM the user enters the service-name at the console prompt. This service-name represents a served-disk on the server. A served-disk is a CD ROM that has been made available by the server to the clients on the network.

From the viewpoint of the console environment a server consists of one or multiple disks. Each disk on the server is uniquely defined on that server and a server is uniquely defined in the network by its Hardware Ethernet address. A server also has a name associated with it. The name of the server is assigned by the user/manager and it is up to that person to manage the name space issues of servers. (That person or persons are responsible for the uniqueness or non-uniqueness of all the server names on their network.)

The server design document contains a complete description of the naming scheme. Details of the scheme necessary for the client are discussed in this document.

The server name is assigned to it when the server is initialized. This name is stored in the server's NVR. The name consists of 1 to 6 characters and should be unique across the network. Future versions of the server may support names up to seventy characters in length. If the name is not unique, then the console output will show that two or more entries with the same service-name exist. The first occurence of the service-name is printed out on the screen per the default format. Subsequent duplications of the name are printed out with numeric indicator, which is a period followed by a number inside of parenthesis. This numeric indicator is a flag to the user that a duplicate service-name exists, and therefore the user must examine the Ethernet address associated with the service-name to determine which of the duplicate serivce-name entries to boot from. For example, if a duplicate service-name was - COMPACT_DISK -, then the first occurence of this name appears as - COMPACT_ DISK - and the second occurence of the name appears as - COMPACT_DISK.(1). To boot from a duplicate name the user must type the service-name followed by the duplicate string identifier - COMPACT_DISK.(1).

The legal characters for a server name are printable ASCII characters. The illegal names are names that cause conflicts. These names include names of existing devices on the client, such as, DUA0, and names that conflict with the duplicate string identifier. An example of a conflicating duplicate string identifier is: .(1). For version 1 of the server, service names are a combination of the server names, and a unit index.

Some examples of legal "server" names are:

```
SERVER
VMS
ULTRIX
```

Some examples of legal "service" names are:

```
SERVER1
VMS1
ULTRIX1
```

The "show services" and "find services "commands have some qualifiers that the console interprets and takes action on. The qualifiers are:

- /Full
- /XMI
- /BI

For the default command, with no qualifiers, the name of each service (served disk) is output to the console. The default syntax is the same for the show service and find service command. Only the show service command is shown below.

```
        >>> SHOW SERVICE

The default output format is:

        Service-Name
        Ethernet Address
        Service_description_string

The output format for duplicates is:

        Service-Name.(#)
        Ethernet Address
        Server-Name
        Service_description_string
```

Each service-name output entry is separated by a blank string. The output on the console terminal for two services is:

```
        Service-Name
        Ethernet Address
        Service_description_string

        Service-Name
        Ethernet Address
        Service_description_string
```

The service-name is the name of the service on the server box. If the service-name is a duplicate of another service already found in the database, then a number follows the Service-name indicating that it is a duplicate. The Service description string is an ASCII string that the server has stored in it that relates to that service. The Hardware Ethernet address is the address of the server box. With this option the console program uses the first Ethernet controller it finds and performs its queries on that unit. A boot command subsequently issued by the user will default to using this same controller. The user has the option of specifying another controller with the /XMI, and /BI qualifiers on the input command line. The syntax for this command is:

```
>>> SHOW SERVICES/XMI:2/BI:3

The output format is:

 XMI:#   BI:#

        Service-Name
        Ethernet Address
        Service_description_string
```

The user is allowed to specify both XMI and BI numbers as well as either the XMI or the BI number. If the user specifies only the XMI the default is to look for the first Ethernet controller on the BUS specified and use that controller. For example, if the user specifies XMI:4, but no BI value, the console program looks for an Ethernet controller the specified XMI bus.

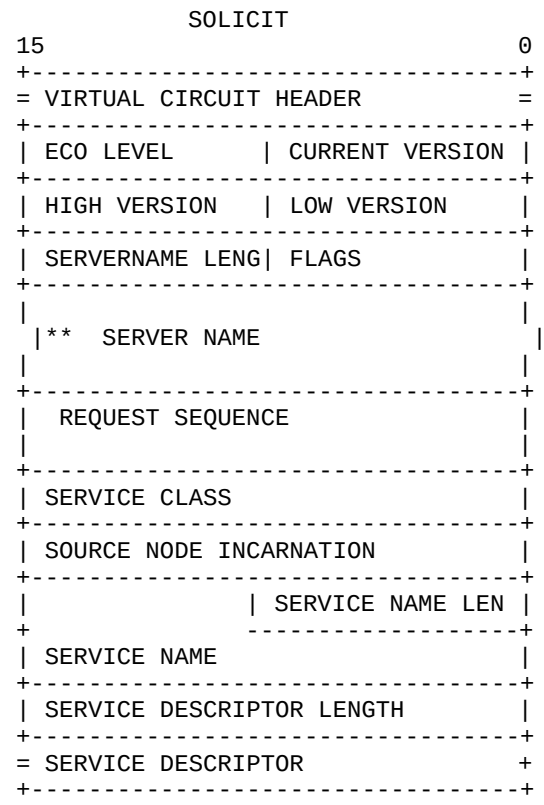
The user obtains more information on each server with the /full qualifer. The syntax and output for this option is:

```
>>>  SHOW SERVICES/FULL

     or

>>> SHOW SERVICES/FULL/XMI:#/BI:#
```

The output format is:

```
 XMI:#  BI:#

        Service-Name
        Ethernet Address
        Server Name
        Service_description_string
        Current Supported Protocol Version, ECO_level
        Service_rating, Service Class
        Unit Number of Disk on Server, Boot Device Type
```

The client CPU obtains this information by issuing a LAST solicit command. The format of the LAST command is:

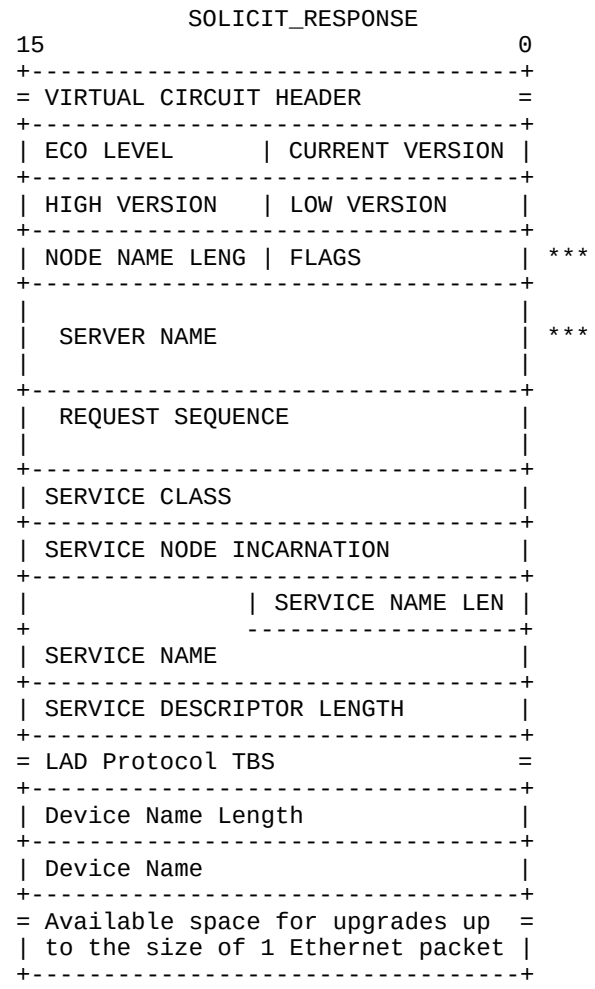


** This field is fixed at sixteen bytes. We are working on getting it changed to a variable length field with a longword size field, and a maximum size of 255 bytes.

The macro template header for this message appears below.

```
;+
; SOLICIT message
;-
CURRENT_VERSION:          .BLKB 1  ; unsigned protocol version of the message
ECO_LEVEL:                .BLKB 1  ; unsigned engineering change order
LOW_VERSION:              .BLKB 1  ; lowest version support by this server
HIGH_VERSION:             .BLKB 1  ; highest version supported by this server
FLAGS:                    .BLKB 1  ; bit 0 = 1 if transport is master
                                   ; bit 1 = 1 if transport is slave
                                   ; bit 2 = 1 purge all paths
                                   ; bits 3-7 must be zero
SERVER_NAME_LENGTH:       .BLKB 1  ; length of server name length
SERVER_NAME:              .BLKB 16 ; server name ** fixed by last protocol
REQUEST_SEQUENCE:         .BLKL 1  ; transport overhead
SERVICE_CLASS:            .BLKW 1  ; set to zero on inquiry for service
SERVICE_RATING:           .BLKW 1  ; service quality
SOURCE_NODE_INCARNATION:  .BLKW 1  ; value indicating we started something
SERVICE_NAME_LENGTH:      .BLKB 1  ; length of data to follow for first disk
SERVICE_NAME:             .BLKB 16 ; name of served disk
```

Each server responds to this received message with a LAST solicit-response message. This response message contains a description of each service (served-disk) on the server. The client creates a database of the responses it receives, saving off the information it needs on each served-disk, with the goal of accessing the disk. The response message from the server contains the information below:



*** This field is fixed at sixteen bytes. We are working on getting it changed to a variable length field with a longword size field, and a maximum size of 255 bytes.

The console program is responsible for determining the controller port address used to send and receive messages. The console program either picks up that information from the input typed by the user, or determines that information from its search for an Ethernet controller board. This information is later passed to the other drivers during the booting process.

The functions of the console program for this command are described below. The console:

- Parses the command input
- Finds the information it needs for the controller, e.g. XMI number, and BI number
- Calls the ACBTDRIVR initialization routine with the information the driver needs:

    function qualifier indicating show services command is to be performed
    XMI/BI adapter address
    location of Ethernet boot driver
    Base of good memory to deposit blocks from server

ACBTDRIVR initilizes the appropriate data and sets up any of its data structures. This includes loading the input parameters into its local data area. It then proceeds to set up the calling interface for the console's Ethernet boot driver. After the Ethernet boot driver returns, the ACBTDRIVR checks the status return and returns to the console program. The console then checks the status return and if successful it calls the operational portion of ACBTDRIVR with the following input:

- Port Information Field (XMI and BI location)
- Time Elapsed Value if any
- Pointer Count Area
- Starting Location to Store Data
- Maximum Location Size of Data Area
- Maximum Time to Return if any

The PORT INFORMATION field is the port the driver is to use to query servers. The default is the first Ethernet device found by the console.

The TIME ELAPSED VALUE is a debugging value for development. This field is not advertised to the user. It represents the time to wait for "any" input back on the wire. After this time the boot driver is to return to the console and report back that not a single response was received. This value is specified with the qualifer "/elapsed".

The POINTER COUNT area is a longword location that contains the number of server entries in the server database. The simplest method is to zero this location out on entry and increment it once each server entry has been made. It is assumed that each server entry is stored contiguously and not held in any queue structure. This assumption does not preclude the implementor from using such as structure; however, in the concern for space, simplicity is used.

The STARTING LOCATION TO STORE DATA represents the data area where the database information is stored. ACBTDRIVR manages this space. It zeros this location before issuing a solicit message to the network. It increments the value in this field everytime it adds an entry to the database, up to the limit of possible entries.

The MAXIMUM LOCATION SIZE OF DATA AREA is the maximum area that can be written by the boot driver. This value ensures that the boot driver (ACBTDRIVR) does not overwrite any other data. This value can be infinite, designated by a -1.

The MAXIMUM TIME BEFORE RETURN is the maximum amount of time that the boot driver is to collect data. This parameter is used for debugging of the drivers. If this value is not specified the server uses a default. This field is specified by using the "/maxtime" qualifer on the command. This value ensures that the ACBTDRIVR returns to the console program, thereby allowing the process to continue and not be hung.

This information is passed:

- Number of parameters
- IO request for a solicit
- Time Elapsed Value
- Pointer to Count of # of Servers
- Starting Location to Store Data
- Maximum Location Size of Data Area
- Maximum Time to Return

The ACBTDRIVR takes this data and takes the actions indicated. The IO REQUEST field can be either a read request, a solicit, a request for a specific service, or output all servers available. The IO qualifiers for these calls are:

- IO$_READLBCK for a read request.
- IO$_ACCESS for a general solicit of all services.
- IO$_CREATE for a request to a specific server to start and connect to the server.
- IO$_NETCONTROL for outputting the server entries in the database.

For the show services command, the I/O qualifer passed to the ACBTDRIVR in this case is the IO$_ACCESS qualifier. It stores all relevant information in the appropriate area and initialies any necessary data. One such area that is initialized is the data area and pointers to the services offered. Once the initialization is complete control is transferred to an area that handles this request. In the case of the IO$_ACCESS request, the driver makes a LAST message that requests all servers to provide information. This LAST message is called a Solicit message. It then creates a calling interface that the Ethernet boot driver understands. The Ethernet boot driver is called with a request to send a multicast block requesting service over the net. After it sends the message it returns back to the ACBTDRIVR. The ACBTDRIVR then sends repeated requests to the Ethernet boot driver to read in blocks. All servers, once they have received a solicit message, return their server information to the source of this solicit message. The format of the return is:

The ACBTDRIVR continues to take information in until either time has elapsed or all space is taken up. Time can either elapse from the Ethernet boot driver receiving no messages in the time alloted, or the total time to receive messages having expired. The space can be taken up by the ACBTDRIVR having received the maximum messages it can store.

The ACBTDRIVR receives a message back from the Ethernet boot driver and starts processing it. It strips off any headers and data that are not relevant. Next the pointers and count fields are updated to indicate that another entry in the table has been made.

After an entry has been made, the ACBTDRIVR checks to see if it should return to the console. TBD, duplicates are tested for by comparing Ethernet addresses. It should return to the console under the conditions listed below:

- Errors returned from the Ethernet boot driver
- Errors returned from the ACBTDRIVR
- Maximum number of messages received
- Maximum alloted time to collect messages
- Maximum time waited to collect a message

If none of the conditions above have happened the ACBTDRIVR continues to collect data. Otherwise ACBTDRIVR returns with a status condition. Out of the above conditions, the following ones return errors:

- Errors returned from the Ethernet boot driver
- Errors returned from the ACBTDRIVR

Successful returns are:

- Maximum number of messages received
- Maximum alloted time to collect messages

Returns with warnings are:

- Maximum time waited to collect a message with no messages being received.

In the latter case it is a warning to indicate that no servers are responding or available.

Upon a successful return status from the ACBTDRIVR, the table has been loaded with server entry names, and updated with pointers to indicate the number of entries. Control now returns to the console. The console checks the return status, and if all is successful it calls the ACBTDRIVR with the IO$_NETCONTROL qualifer. This causes the ACBTDRIVR to read the server database and format the output. To do this the ACBTDRIVR checks to see that it is running in physical mode, with mapping turned off. If mapping is not turned off a warning is displayed. This warning message requires further investigation as to when the PR_MAPEN register is changed after a crash. The check for the mapping status is made to ensure that console IO can be preformed. At present the usage of the console IO routines from the ACBTDRIVR is limited to physical mode only. The restriction can be lifted after sufficient work has been done to guarantee that output appears when running in virtual mode.

ACBTDRIVR picks up the IO$_NETCONTROL qualifier and examines the argument pointer for other variables passed in. In particular it examines for the qualifier:

```
/ full
```

If the /full qualifer has been specied, ACBTDRIVR outputs the data in the full format. Otherwise, just the server names and a short description are output. Note, the short description may consist of all blanks. Note also, that the ACBTDRIVR calls the console IO routines to handle the output. (ACBTDRIVR is responsible or formatting the calls.)

Below is a break down of the actions in this scenario.

```
SHOW SERVICES
Begin
    Console:
             Finds Ethernet address of Client CPU
             Finds port used or to use from the input command
             Loads fields in local structures and registers
             Calls drivers to initialize port

    Console:
             Calls ACBTDRIVR to create a request for servers

    ACBTDRIVR:
             Creates a solicit message and sends it

    Servers/LAST sees a solicit_request message
             LAST answers solicit message
             Message packet gives SERVER_NAME_DISK
                  SERVER_NAME
                  DISK #
                  ETHERNET PATH INFO

     Console:
             Has boot drivers reading solicit_responses
             ACBTDRIVR returns to Console with status
             Status is checked, process continues upon success
             Calls ACBTDRIVR to output database
             ACBTDRIVR outputs database.
```

2.4.1.2 BOOTING OPERATIONS AND SYNTAX

Booting consists of the user typing in the boot command, the console program parsing the command, and calling the class boot driver to read in the boot block program. The boot block program uses the class boot driver to read in VMB. After VMB is read in control is transferred to it and VMB reads in the next module. For VMS this module is sysboot. Sysboot continues on reading in the operating system, turns on virtual mode addressing, and then transfers control to the next module that operates with virtual addressing mode on. The boot drivers prior to VMB reside in the console. VMB uses the boot drivers read in with it. The class driver read in with VMB has the capability to do read requests. It does not have the support for the show server command.

The actions taken to boot are:

The user enters a boot command. The syntax for the boot command is:

```
>>> b service_name
```

The optional qualifiers for this command are:

- software boot flags, /R5:#

- Port information: /XMI:#, /BI:#

The console program parses the input and determines that the user is booting from a server. It next determines if the client server database has any entries. If the database is empty it informs the user of this fact and issues a prompt:

```
"NO SERVICES, PERFORM FIND SERVICES? (Y)"
```

The goal of the prompt is to inform the user that the database has no server entry in it, and to let the user perform a show services command to find the service.

When the database is empty the console program issues calls to the class boot driver to fill the database. The class boot driver posts some read requests to the port boot driver, and then requests the port boot driver to write a solicit message over the wire. Responses from the solicit message are received and loaded into memory.

If and when the database is not empty, the console searches the database for an entry matching the server name. The entries in the database are referred to as Available Target Identifiers, ATI's. If a matching entry is not found, the console sends a message back to the user indicating "no such server". The user then has the option to perform another show service command to find another server to boot from. When a matching ATI entry is found in the database, the console moves the ATI to a portion of memory that it has reserved for it. This portion of memory is located before VMB and is not used or overwritten by VMB. The console then loads the necessary information from the ATI into the calling interface locations. The register R3, usually denotes the device unit number, is used to point to the Available Target Idendifier. The console program picks up the destination Ethernet address, port information (XMI,BI), boot device type, and the served-disk name.

The console program now wants to read in the boot block, LBN 0, sized at 512 bytes. To do this it calls ACBTDRIVR with a request to set up a circuit to the served-disk. ACBT-DRIVR creates a start message and hands it off to the Ethernet port driver to send. The Ethernet port driver waits for a response and returns to the ACBTDRIVER with either the response, or a status indicating that no response has occurred in the time allowed for a server to respond. On all failure cases, bad status is returned to the console program which is responsible for outputting failure messag es to the console device. Upon succesful return status from the Ethernet port drivers sending of the start message, ACBTDRIVR gives a connection request message to the Ethernet port driver to send. The Ethernet port driver sends the message and waits for response or a time expiration. Upon success, ACBTDRIVR returns to the console program for the next sequence in the booting operation.

The console program now begins the reading phase. It calls ACBTDRIVR with a logical block read request, staring at block 0, with a size of 512 bytes. ACBTDRIVR creates a LAST/LAD read message. It then calls the Ethernet port driver to sends this message to the served-disk and waits for a response. The target server handles the request and sends data back to the client. The client Ethernet driver receives the blocks of data it has requested and returns to the ACBTDRIVR. The ACBTDRIVR strips off the overhead and loads/reloads only the 512 bytes requested into the location requested.

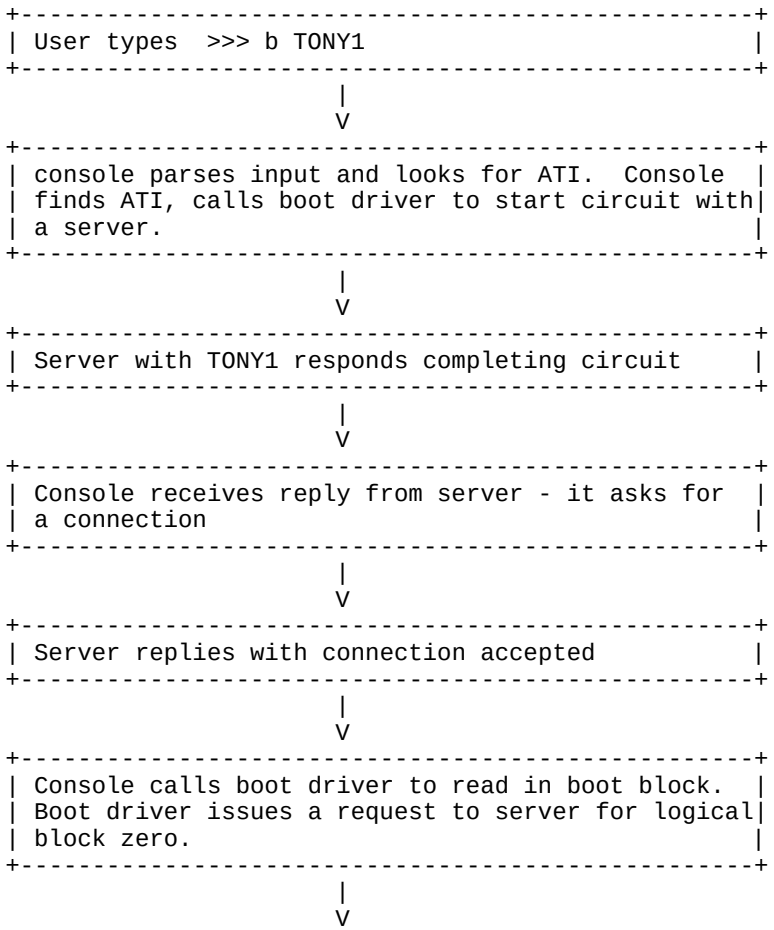
After the boot block has been read in the console program sets up the calling interface to the boot block program. It sets up any registers necessary for the boot block program to call back into the class boot driver (ACBTDRIVR). Control is then transferred to the boot block program.

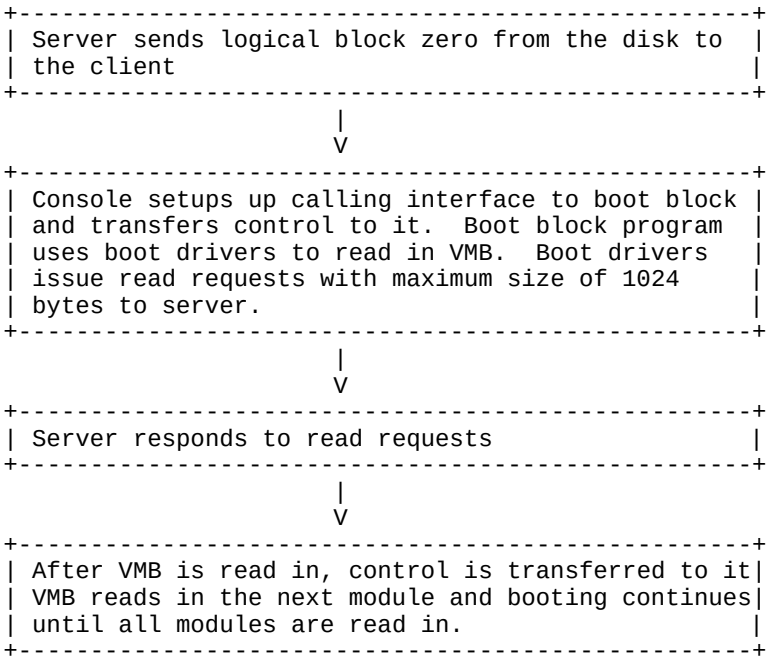
The Boot block program continues to use the console Ethernet boot driver, and the ACBT-DRIVR in the console until VMB has been read in. VMB is read in by the boot block program issuing read requests to the ACBTDRIVR. The ACBTDRIVR issues read packets, with the restriction that the maximum number of blocks to be read in on one read is 1024 bytes, to the server through the Ethernet boot driver. This restriction of 1024 bytes per read request is necessary to avoid the necessity of having to re-order the received packets. With requests of 1024 bytes, all packets received by the Ethernet boot driver are in order as long as only one read request for blocks of data is outstanding at a time.

Once VMB has been read in the console sets up the calling interface to VMB and control is transferred to VMB. VMB contains Ethernet boot drivers and its version of the LAST /LAD protocol boot driver, named BEBTDRIVR. BEBTDRIVR supports read requests to the served-disk. It handles maximum transfer sizes of 1024. So when it receives a request from VMB for a transfer of a larger size, BEBTDRIVR must break the request up into multiple requests, and issues those requests one at a time, waiting until the data from the previous request has been received. This restriction is not necessary if the programmer chooses to handle the re-ordering of out-of-order blocks of data being returned to it.

The boot drivers in VMB are finished with their task once all initial images are read in, IPL is lowered to an operating system level, and an operating system exists that has support for operating system device drivers.

Below is a flow of code for the operations just described.





2.4.1.3 VMB and other Elements

VMB determines what device to use for booting from the boot device type symbol, and then relocates the boot driver associated with this device. All of these actions are taken under the assumption that VMB is interacting with one driver. The device types supported by NI-CD ROM are:

```
BTD$K_SERVER_DEBNA
BTD$K_SERVER_DEBNI
BTD$K_SERVER_XNA
```

The port/class drivers are two different drivers that act as one driver. There is only one class driver for NI-CD ROM but there are possibly multiple types of port drivers. VMB issues calls to the class driver only, never to the port driver. VMB finds the class driver through the boot device type symbol; valid device names for servers are a concatenation of the server designator and the Ethernet designator. The Ethernet designator is used by the class boot driver to determine which port driver it must use. Because of this port/class boot driver architecture it is necessary for VMB to relocate both the port and class boot driver; however, the VMB code is implicitly performing this action. VMB performs the relocation of both the port and class boot drivers by calling the "action" routine of the class boot driver. The action routine does the following:

- determines which port driver to use,
- finds the location of the port driver,
- relocates the port driver behind the class driver,
- increases the class driver size to include the port driver,
- zeros out the address of its own action routine it the device table

- finds the boot driver table for the port driver and stores the offsets to those entry points (The class driver uses the offsets into the port driver's routines when it calls into the port driver.)

VMB will now relocate both the class driver and port driver as if it were one driver.

The class driver is responsible for creating request messages, packeting the up, and delivering those packets to the Ethernet boot driver. The class driver is also responsible for receiving packets, unpacking them, and relocating the blocks of data received to the correct location in memory. The Ethernet boot driver is responsible for sending and receiving packets over the wire to and from the destination server. The class driver's initilization routine calls the initialization sequence for Ethernet driver. The class driver must set up all appropriate registers and fields in the calling interface to the Ethernet driver. The Ethernet driver must return to the class driver with a status message.

After initialization calls are made to BEBTDRIVR through the BOO$QIO calling interface. The actions on a read request are described below.

BEBTDRIVR receives a request to read a block of data. It creates a command data message.

- leaves room for virtual circuit header
- leaves an open run message header
- Fills in all fields.
- The transaction slot number is incremented and placed into the field.
- The sequence number is set to 1, unless a retry
- Max segment number is 1
- Current segment number is 1
- Command response timer is set to 5 minutes (debug for delta)
- Transaction response timer is set to 10 minutes (debug for delta)

Next is the format used to query the server:

- disk #
- starting LBN
- Number of bytes desired

RESPONSE DATA

The server responds with the response format.

The format is pretty much the same except for the data and data count field. The data count contains the number of bytes returned plus other information for debug possibly.

- Disk # data retrieved from
- starting LBN
- Number of bytes
- desired,

- Actual data

For each request to read a block the actions below take place. BEBTDRIVR:

- creates a packet to send to the server
- requests the Ethernet boot driver send that packet
- waits for status
- acts upon status
- If status was a success it issues a read request
- to the Ethernet boot driver
- If status was a failure it exits

The BEBTDRIVR continues to issue reads until all packets for a single request have been read. When all requests have been read they are processed and returned to the caller.

2.5 Component Design Criteria and Constraints

The performance of the client software must be reliable. The user must be able to determine the progress of the event taking place.

The client software must not crash during the initial software installation phase. During the installation of software the client must also not crash. The booting software must have a predicatable behavior. It must not hang inifinitely. It must have a reliable and deterministic behavior that the user can depend on and predict. The software must be designed so that when booting the driver succeeds in booting successfully, or in the event of a problem, it makes the determination that it cannot succeed and reports back to the user with at status. evolvability A port/class design was used to divide the transport architecture as much apart as possible from the actual mechinism doing the transport. As little code as possible was put into the ROMs on the machine. This is because of the limited space available and also to keep the amount of code that has to be maintained and pathced on the rom minimal.

2.6 Component Test Specification

The unit test on the client consists of success tests and failure tests. Success is defined as the booting of the machine from the server. Failure cases are many if not inifinite. The highly probable failure cases that are to be tested are:

- Server losing power and not rebooting
- Server losing power and rebooting
- Server having the booting disk removed
- Server having another disk removed
- Server having bad media in it, media is only partially readable by the head
- Server having a bad distribution media on it.
- Not all parts of the kit made it to the media

- Error Insertion

2.7 Component Environment

The boot driver will follow the standard rules for re-initialization.

2.8 Error Handling

The errors for the code in the console level and the VMB level are different. The console handles the determination if any servers are out on the network. In the case that no servers are on the network the console returns the an error indicating no servers.

APPENDIX A

OUTSTANDING ISSUES

ISSUE: find out from mariah what is available in memory speed. ISSUE: what is the protocol version and ECO version we need, Marshall Goldberg.