Applied
Microsystems
Corporation

NetROM™

User's Manual

December 1995
P/N 922-07000-00

Contents

Chapter 3
Installation

Chapter 4
User Interface

Chapter 5
Debugger Support

Chapter 6
Alternate NetROM Interfaces

Chapter 7
Emulation Memory Mailbox Protocols

Chapter 1
Introduction

NetROM is a powerful communications and ROM-emulation tool for use in embedded-systems design. It enhances the embedded-systems development environment in ways that increase productivity and decrease development time and cost.

NetROM facilities communication between a host computer and a target system. Using the high-speed data-transfer rates available on Ethernet LANs, NetROM updates emulation memory with new images much more quickly than conventional serial- or parallel-link ROM emulators.

NetROM gives developers convenient communications paths to target systems via a serial link and a mailbox system in emulation memory. The mailbox system is particularly useful for target systems that do not have serial ports. You can also use the mailbox system to give targets not capable of writing to ROM addresses write capability.

NetROM can act as a communications nexus, collecting messages from the target and sending them over the Ethernet to the user, and collecting messages from the user and forwarding them to the target. These communications paths can be interactive sessions or they may be data-packet transfers between the target system and a remote host program.

NetROM provides a set of eight status inputs that can be connected to any signal on the target system and sampled as desired. NetROM also provides eight command signals that can be connected to the target and asserted by the NetROM user.

NetROM uses standard Internet protocols such as BOOTP, RARP, TFTP, and TELNET. NetROM is network-manageable using SNMP.

NetROM Features

NetROM is a versatile development tool that can be adapted for most embedded systems configurations. The following are NetROM's principal features:

ROM Emulation

- 1 MB (megabyte) of emulation SRAM. Can break SRAM into 4 pods, each emulating a maximum of 256 Kbytes, or 2 pods, each emulating up to 512 Kbytes.
- Support for 8-, 16-, and 32-bit words by clustering pods into groups.
- Support for 64-bit words by using more than one NetROM unit.
- Support for more than 1MB of emulation by using more than one NetROM unit.
- Emulation of 64K, 256K, 512K, 1 Mbit, 2Mbit, and 4Mbit ROMs.
- Emulation of larger than 1 MB ROM by using more than one NetROM unit.
- Simultaneous emulation of multiple ROM types and word sizes by using different pods.
- 85 ns response time emulation memory and 55 ns for NetROM 450.
- 28-, 32-, and 40-pin DIPs, and 32- and 44-pin PLCC ROM socket pods.

Communications

- Fast emulation downloads over Ethernet, using standard protocols, such as TFTP or TCP.
- Communication with target systems using RS-232; and emulation memory mailboxes.
- Address resolution using BOOTP or RARP.
- Eight status signals from the target that can be polled at will.

- Eight target command signals that the user can assert.
- Network manageable by SNMP.

User Interface

- Multiple user sessions with NetROM and/or the target, using standard protocols such as TELNET.
- Robust command-line interface.

Integration with Debuggers

- Support for passing data between a debugger running on a remote host and the target system.
- Extended debugger support for updating the downloaded image, resetting the target, and similar features.
- Emulation memory writable by the target system even if target hardware does not allow it.
- Downloadable RAM modules to support optional features like JTAG debugging and Virtual Ethernet.

Embedded Systems Development Environment

Embedded systems are specialized microprocessor-controlled devices, of varying sizes, used for specific purposes. Examples range from PC boards, network switching devices, and laser printers to microwave ovens and the computerized controls in a car. The NetROM device itself is an embedded system.

In most development environments for embedded systems, there are four main components:

1. An embedded system under development—the *target*.
2. A computer used to develop the embedded system—the *host*.
3. A communications path between the host and the target.
4. A ROM emulator or ICE.

Although NetROM's ROM emulation features are powerful, NetROM's most important function is communication between the development host and the target system. In general, there are three communication-path types between the target system and the host computer: download, console, and debug.

These three paths are common aspects of embedded systems development, and, usually, each path type has to be implemented using a separate tool.

NetROM, however, gives developers a single tool capable of implementing all three paths from the host system to the target. The three communication paths are discussed in detail in Chapter 2; Table 1-1 briefly describes each path type.

Table 1-1 NetROM Communications Paths

Path	Description
Download	Mechanism in which an image file created on the host system becomes accessible on the target system.
Console	Mechanism in which the user can communicate with the target system.
Debug	Mechanism in which an embedded systems debugger program communicates between the host system and the target system.

Standard Development Environment

Many embedded systems development environments use RS-232 serial lines to implement each of the three communications paths described above. This approach has three main drawbacks: It lacks speed and portability, and it is inconvenient.

Serial communications lines are much slower than LAN technologies like Ethernet. The download path, in particular, is affected because of the large amounts of data that must be transferred. It is not uncommon to download megabyte-sized images into emulators, which obviously can take considerable time over a serial line.

With the variety of host systems available, software to drive serial ports for each different system is sometimes difficult to find. This problem is particularly acute when the host computer is downloading to an emulator that uses a proprietary protocol on top of the RS-232 serial link.

Using serial lines frequently can be complicated to set up and to use. To physically set up a suitable environment for debugging, the engineer must locate computers with serial ports, software drivers for the serial ports, then connect cables between the target system, the ROM emulator, and the host computer. This process may require the host computer with the serial port run a debugger, terminal emulator, ROM emulator download, or all three.

Development Environments Using NetROM

NetROM technology addresses the drawbacks of a traditional RS-232 development environment. In addition to emulating ROMs, NetROM functions as a communications nexus between the host computer and the target system. Because NetROM connects to a high-speed LAN, it can multiplex all of the different communications paths from their respective sources onto the network.

NetROM is fast, because LANs are fast. This allows downloads to be completed much more quickly than is possible with serial lines.

NetROM is portable. With the variety of host systems available, finding the software to drive serial ports for each is difficult. The problem is acute when the host computer downloads to an emulator that uses a proprietary protocol on top of the RS-232 serial link. With NetROM you do not have to modify serial drivers.

NetROM is convenient. The debug path from the host computer to the target can go through NetROM, with the host-side program using standard TCP interfaces, such as sockets. Physical setup of the debug environment is simplified, because serial connections can go directly from NetROM to the target. The emulation pods themselves provide non-RS-232 forms of message passing, and NetROM allows supplementary control of the target using the command and status signals.

Documentation Overview

This manual contains detailed information about the NetROM product, its services, installation, user interface, debugger support, alternate interfaces, emulation memory and more. It also includes complete reference guides to the NetROM commands and variables.

The manual is organized as follows:

Chapter 2. NetROM Services
Discusses the NetROM embedded systems development environment, including its implementation of the three communications paths. Also describes the NetROM console, command and status signals, and LEDs.

Chapter 3. Installation
Gives step-by-step instructions for the installing the NetROM hardware and software, including how to get started.

Chapter 4. User Interface
Describes the NetROM user interface, including the command set and environment variables.

Chapter 5. Debugger Support
Describes NetROM's debugger support features.

Chapter 6. Alternate NetROM Interfaces
Describes how users can write their own programs to interface to NetROM.

Chapter 7. Emulation Memory Mailbox Protocols
Explains emulation memory mailbox protocols for targets that cannot write to their own ROM space.

Chapter 8. Virtual Ethernet
Introduces Virtual Ethernet, an optional downloadable RAM module to NetROM, including instructions for installation.

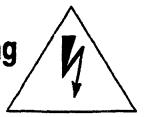
A set of appendices, a glossary, and an index follow these chapters. The *NetROM Installation Notes* and the *NetROM Hardware Interface Reference* are included at the end of this manual.

Documentation Conventions

This manual uses the following conventions:

- Book titles, emphasized words, command names, and keywords are in *italics.*
- Command parameters are in **boldface.**
- Computer programs are in `constant-spaced font.`
- Environment variable names are in "quotation marks."
- Items that are optional are enclosed in [square braces].
- Items that are mutually exclusive are separated by a vertical bar |.
- Mutually exclusive items, one of which is mandatory, are enclosed in {braces}.

Warnings, Cautions, Notes

Warning 

Warning messages appear before procedures and alert you to the danger of personal injury which may result unless certain precautions are observed.

Caution 

Caution messages appear before procedures and indicate that damage may be done to the emulator or to your target system unless certain steps are observed.

Note 

Notes indicate important information for the proper operation and installation of your emulator.

Support Services

Applied Microsystems provides a full range of support services. The NetROM is covered by a 90-day warranty that includes full applications phone support. Additional support agreements are available to extend the initial warranty and to provide additional services.

If you have trouble installing or using the product, consult your manuals to verify that you are following the correct procedures.

If the problem persists, call Customer Support. Customers outside the United States should contact their sales representative or local Applied Microsystems office. When you contact Customer Support, have your serial number available.

Telephone

800-ASK-4AMC (800-275-4262)

(206) 882-2000 (in Washington State or from Canada)

Internet address

If you have access to the Internet, you can contact Applied Microsystems Customer Support using the following address:

support@amc.com

You can also browse the Applied Microsystems World Wide Web page using the following URL:

http://www.amc.com

FAX

(206) 883-3049

Chapter 2
NetROM Services

This chapter provides a general description of NetROM's implementation of the three communications paths introduced in Chapter 1. It also discusses the NetROM console, NetROM's command and status signals, and the NetROM LEDs and their uses.

Communications Paths

The embedded systems design, there are three NetROM communications paths from the host to the target system:

Download path

How new images are loaded into NetROM's emulation memory. This path allows the target system to respond quickly to changes made in the source code residing on the host computer and reduces the cycle time between modifying code and testing the modification.

Console path

How humans interact with the target system; i.e., how they configure it, control it, and monitor it. This usually consists of a serial port running a simple terminal emulator that processes commands. This path allows the development engineer to easily inspect and monitor the system for bugs or unexpected behavior.

Debug path

How the debugger running on a host computer communicates with the target system. This allows the engineer to set breakpoints, examine registers and data, and respond to error conditions almost as if the target system were a program running on the host computer.

NetROM Download Path

The download path from the host system to the target is used to transfer ROM images from the host system, where they are developed, to the target, where they are used. These images are commonly executable code, but can contain other data as well; for example, some target systems may use ROMs to store graphics or configuration information. Images are generally developed on the host system using a compiler or some similar software tool.

Pods and Pod Groups

NetROM has one megabyte of emulation SRAM. This is broken down into four partitions, each of which can be used to emulate a single ROM. The maximum size of each partition is 256 Kilobytes, the size of a 27c020 ROM. These partitions of emulation memory connect to the target system using ribbon cables which end in the type of connector used for the particular type of ROM being emulated. We will refer to the partitions of emulation memory and their associated connector cable as emulation pods, and the ROM socket connectors on the end as plugs. The term "pod" will be used interchangeably for the emulation memory itself, or for the connector cable and its plug. The two most common types of plugs are DIP (Dual In-line Package) and PLCC (Plastic Leaded Chip Carrier). DIP plugs are rectangular and PLCC plugs are square. Note that since the cables and plugs can be detached from NetROM, the same type of ROM, e.g. a 27c020, can be emulated whether it comes in a DIP or PLCC package for a particular target system.

NetROM pods are numbered 0 to 3; each is capable of emulating a single 8-bit ROM containing up to 256 Kilobytes. To make using pods simpler, pods can be grouped together to form pod groups. Pod groups are one or more 8-bit pods combined to make 8-, 16-, or 32-bit words. Each pod is considered a ROM by the target system. Pods can be combined to create words wider than eight bits; for example, a target with two ROMs might use them to make 16-bit words, where one pod is byte 0 of the word and the other is byte 1. We will refer to this as a *parallel* pod organization. Pods can also be combined, not to create *wider* ROM words, but to create *more* words. For example, a target with two ROMs might use them to create twice as many consecutive 8-bit words as it could have with only one. We will refer to this as the *serial* organization of pods. Serial and parallel organizations are not mutually exclusive; a target with four ROMs might use them to create twice as many 16-bit words as it could with only two. See "podgroup" on page 4-116 for detailed information on pod groups. Pod groups have five defining qualities:

1. ROM type:

   All pods in a pod group emulate the same type of ROM. This is because system designers use groups of 8-bit ROMs, all of the same type, to implement ROM memories on their embedded systems. Thus, it is not necessary to specify ROM type for each pod in a pod group, but only for the pod group as a whole.

2. Word size:

   This determines the number of emulation pods operating in parallel to create 8-, 16-, or 32-bit words. That the pods are read as words is an artifact of the hardware design of the target system. The target will generally have an address which is the start of ROM space and will read whole words from that address. For example, 2 pods may emulate 27c020 ROMs in parallel to create a ROM space with 256 Kwords, where each word is 16 bits, not eight bits, wide. Word size is closely tied to ROM count, described below.

3. ROM count:

   This is the total number of pods participating in emulation. If the ROM count is greater than the word size divided by 8, then some of the pods used in emulation are being used serially to create a ROM address space longer than is possible with a single set of pods operating in parallel. For example, four pods may emulate 27c020 ROMs using a word size of 16 bits. In this case, the word size divided by 8 is 2, so the pods are being used by the target as two sets of parallel ROMs operating in serial to provide a longer address space. This space is 512 Kwords long, where each word is 16 bits wide. shows the valid combinations of ROM count and word size.

Table 2-1 Combinations of ROM Count and Word Size

Word Size	Number of ROMs Emulated			
	1	2	3	4
8	Yes	Yes	Yes	Yes
16	No	Yes	No	Yes
32	No	No	No	Yes

4. Pod order:

   For target systems which use multiple pods, in serial or in parallel, it may be desirable to specify the correspondence between emulation pods and ROM bytes. Target systems generally number the ROM sockets on the embedded system. For example, a 16 bit target might number its sockets 0 and 1, in accordance with which byte is within the 16-bit word. The engineer debugging this target might want NetROM pod 0 to byte 0 on the target and pod 1 to be byte 1, or might prefer that pod 1 be byte 0 and pod 0 be byte 1. Consult "podorder" on page 4-117 for more information.

5. Writability:

   The system engineer may want some pod groups to be writable by the target system. This allows the target to set breakpoints in a ROM image, or to modify the image in other ways. Pod groups by default are read-only, and any attempt to write them is quietly ignored. The writability attribute controls both write cycles which use the emulation pod's write line and the external write line. Note read-only targets can request that NetROM write emulation memory for them (see "Setting Emulation Memory" on page 7-21).

The pod memory (emulation memory) may be configured to emulate either flash ROM or static RAM. The difference being how NetROM reacts to the WR signal when OE is asserted. The environment variable *writemode* control s this environment. The default mode is FLASH.

In addition, pod groups have two other characteristics, both of which exist for the convenience of the development engineer:

1. Group name:

   Pod groups can be assigned a name, so that when engineers examine the pod configuration for NetROM, they will not have to remember which group is being used for what purpose. Names can be any mnemonic, and are optional.

2. Target address:

   This is the 32-bit address indicating the start of the pod group in the target's address space. The ability to specify a target-side starting address allows engineers to compare the contents of emulation memory directly with the binary image created on the host system, using the contents of a map file for addressing. Target addresses default to 0, but can be set to any 32-bit value.

Each emulation pod can be used independently or as part of a pod group. Since there are four emulation pods, there are a maximum of four pod groups available. As described above, each pod group corresponds a ROM address space from the point of view of the target. For example, pods 0 and 1 can be used to emulate 27c020 ROMs making up ROM space of 256

Kwords, where each word is 16 bits. Simultaneously, pod 2 can emulate an 8-bit 27c64 containing system configuration information and pod 3 can emulate an 8-bit 27c010 containing graphics tables.

Configuring Pod Groups

In the simplest case, in which the NetROM user simply wants to emulate one pod group, configuring the pod group is quite simple. NetROM allows the user to specify the ROM type, the word size, and the ROM count for the default pod group (group 0) simply by stating his or her preferences on the command line. The pod order (mapping between emulation pods and target system ROMs) can be set instead of, or in addition to, the ROM count and word size. Consult "podorder" on page 4-117 for details on setting the pod order with a pod group. Pod group configuration can be done as part of a startup batch file. After the pod group has been configured, the group can be loaded with an image and emulation can begin. The commands needed to configure pod groups are described in Chapter 4.

Downloading Pod Groups

Each pod group can be downloaded independently. NetROM takes advantage of LAN speeds to accomplish fast downloads. Downloads can be accomplished in several ways. For many users, downloads can be accomplished most easily using TFTP, a standard Internet file transfer program. It is also possible to download new images using TCP connections to a port on NetROM. For many development environments, such as a set of workstations connected on a LAN, it is easier to write network code to communicate between two nodes on the LAN than it is to write code which controls the serial port on a workstation. Network code is generally more portable than serial control code, in addition to being easier to write.

ROM Type Compatibility

It is possible to have NetROM's ROM emulation appear to fail for a particular ROM type, even though a real ROM of that type works. For example, a target system that works with a 27c010 ROM may not work when NetROM emulates a 27c010 ROM. However, the same image may work if NetROM is emulating a 27c020 ROM instead. This situation is generally a result of the target system's hardware interface to the ROM socket.

The pin specifications for the 27c010 describe certain pins as being "no-connects." This is so that the 27c010 will fit into a 32-pin socket, as will a 27c020. However, on the 27c020 *the same pin is specified as an address pin.* As a result, if the target system actually drives that pin NetROM will treat it as address bit 17, and will respond as if it were a 27c020. (Hardware timing restrictions prevent NetROM from masking address pins based on ROM type.)

When a NetROM user specifies a ROM type, the type is primarily used to locate where downloaded images should be placed in emulation memory. NetROM's hardware "pulls up" unconnected address pins, with the result that 27c010 images need to be loaded in the top half of emulation memory. If the "no-connect" address bit is driven by the target, the target will actually be reading data from the *bottom* half of emulation memory.

There are two ways to resolve the problem posed in our example. One is to modify the emulation cables by physically disconnecting address bit 17. The other is to specify a ROM type of 27c020. In situations where changing the ROM type seems to alleviate a problem in emulation, it is a good idea to consult the hardware specifications for the ROM socket on your target system.

NetROM Console Path

Many embedded systems have a serial port for communicating interactively with a user. Some systems have no such mechanism and debugging is correspondingly more difficult, since the engineer must rely on LED displays and DIP switches or similar methods to determine what the target is doing and give it commands. NetROM provides tools to communicate directly with any of these types of target systems. The "readaddr" and "readwrite" paths use mailboxes in emulation memory to communicate with the target system. The difference between them is the "readaddr" method can be used by targets which cannot write emulation memory; it is slower than the "readwrite" method, but it is absolutely generic with regards to target capabilities.

Regardless of the method of communication between NetROM and the target, the host system always uses TELNET to communicate with NetROM. (Actually, the host system can use a direct TCP link to a port on NetROM, but we shall use TELNET in our discussion since it is available in many development environments. Consult Chapter 6 for information on alternate NetROM interfaces.) Using TELNET for all types of communication with the target means that the development engineer can more or less ignore which specific path is being used to communicate with the target once everything is set up. It also means that the engineer can communicate with the target — via NetROM — from anywhere in the office or lab which provides TELNET. This eliminates the need for VT100 terminals with serial cables sitting near the target system.

Another thing to note is that the target doesn't need to have a serial port in order to provide a console to the engineer. By using the emulation memory, which is shared between NetROM and the target, to pass messages, the target can communicate directly with the engineer during the lab debugging. Another implication of using emulation memory to pass messages is that it requires a minimum of working hardware. That is, even targets with a serial port require hardware which can access and program the port. During the very early phases of system boot, this is not always available and software engineers must

use crude methods for debugging. However, targets which execute out of emulation memory already have everything they need to communicate with the engineer.

The protocol which is used to communicate with NetROM through emulation memory is simple, and generic target-side code is provided with NetROM. This code requires minimal porting and provides character-oriented input and output functions. The protocol is described in more detail in Chapter 7.

NetROM Debug Path

Many development environments for embedded systems use symbolic debuggers. These debuggers actually run on the host system but communicate with the target system, generally using serial lines. The debugger sends messages to the target, generally interrogating or setting the target's register state or memory contents. The target sends messages to the debugger, informing it of breakpoints and exceptions. On the host side, the debugger keeps track of symbol tables, source files, and breakpoint status. On the target side there are generally small routines which perform the operations requested by the host side.

NetROM allows the host side debugger to communicate with the target through any of the paths. However, there are two paths between NetROM and the debugger. One is the debug data path. This is a TCP connection, and data received from the debugger is passed directly to the target. The other is the debug control path. This is also a TCP connection, but it is used to pass control data directly to NetROM, which allows debuggers to perform such tasks as directly reset the target or download a new image. The debug path between NetROM and the target, once again, is independent of the path between NetROM and the debugger. Thus, the debugger does not need to be aware of how data is passed to the target; it merely sends it to NetROM and NetROM takes care of the rest.

Optional Downloadable RAM Modules

Applied Microsystems' downloadable RAM modules are licensed applications packages that extend the existing command set or add functionality to the NetROM product. For example, the Virtual Ethernet module gives target systems the ability to become Ethernet communications devices without requiring that they have Ethernet hardware; and the 29K JTAG module adds standard IEEE 1149.1 JTAG debugging emulation to NetROM's normal communications gateway and ROM emulation functions.

These applications are licensed to individual NetROM units and are linked to a specific NetROM version. That is, when you purchase an XDI downloadable RAM module, it can only be used on one NetROM unit and that unit must be running a compatible version of NetROM software.

The application software media contain the computer file(s) necessary to load the module into the specific NetROM system. Module files are loaded using the *loadmodule* command, which automatically executes TFTP, the file transfer mechanism that downloads the file(s) to your NetROM system. We recommend these file(s) be stored in the same directory as your startup.bat file; however, the file(s) can be placed on the server anywhere to which NetROM has TFTP access.

Detailed installation and usage instructions for the modules are included in the respective module's Application Notes or User Manual.

NetROM Console

The NetROM console consists of a "dumb" terminal connected to the NetROM console port by a serial RS-232 line. This port is always active and can be used even when NetROM does not seem to be accessible by the network. The port serves several uses.

First, it allows NetROM users to access NetROM without using the Ethernet. This is particularly useful during the initial stages of NetROM installation, when NetROM's Ethernet address is not known and therefore cannot be associated with an IP address for use on the Ethernet.

Second, in environments which do not offer address resolution servers, such as BOOTP or RARP, the NetROM console allows users to access NetROM in order to configure its IP address by hand. Normal NetROM operation can then proceed, assuming a TFTP server is present to download images. (It is possible to *send* a file to NetROM for download into emulation memory, rather than having NetROM *request* the file; however, such an environment does not allow "normal usage." Consult Chapter 6 for information on alternate NetROM interfaces.)

Third, NetROM users may occasionally want to monitor traffic between NetROM and the target system on the console path, the debug path, or both. Data passed between NetROM and target can be echoed to the NetROM console if enabled; see the *set consecho* and the *set debugecho* commands for details.

Finally, if a terminal is left attached to the console port, it will receive messages about abnormal events. As with many multitasking and potentially multi-user systems, NetROM uses its serial port to provide a log of diagnostic messages about abnormal events. However, most users will not need to make use of this aspect of the NetROM console.

Command and Status Signals

NetROM provides a set of eight command signals, which can be mapped to arbitrary traces on the target system. Command signals are "active low," which means when asserted (set to "on") the signal is near ground potential. Active low traces are common in many computer systems. There are certain electrical considerations involved in using the command signals. First, the command signal must be connected in such a way that it will not cause a short circuit. NetROM command signals are driven by a GAL 22V10-15 capable of sinking a maximum of 16 milliamps at 0.5V. Command signals are tri-state; that is, when they are not asserted, they are "not connected" to the target. Most TTL signals generally drive small amounts of current (about 10 milliamps). However, if the trace to which the command signal is connected drives a large amount of current, when the command signal is asserted there will be a short circuit which may damage NetROM. The solution is to use a resistor between the command signal and the target trace. Command signals are meant to be used in either open-collector circuits or circuits which drive small amounts of current.

NetROM also provides a set of status signals which also can be connected to arbitrary signal traces on the target system. Status signals simply monitor the status of the traces on the target system. They are interpreted as being active low; that is, if the TTL signal on the target side is low, the status will read as "on," otherwise it will read as "off." The status lines are provided as a convenience to the engineer; they can be interrogated at the command line or mapped to LEDs on the NetROM back panel. There are eight status lines and only four LEDs, so the engineer may either map some LEDs to multiple status signals or not use all of the status signals. It is possible to configure signal-to-LED mappings in such a way that the LED lights up with the signal is high; that is the engineer can specify a high-true mapping as desired. NetROM commands to map status signals to LEDs are described in Chapter 4.

Some of the command and status signals have default semantics. For example, command signal zero (0) is assumed to be connected to the target processor's reset line. If a reset line is connected to the target, the engineer using NetROM can apply a hardware reset to the target remotely; refer to "logout" on page 4-94.

NetROM LEDs

NetROM has two sets of LEDs, the network activity LEDs and the status LEDs. The network activity LEDs are controlled by NetROM's Ethernet interface, but the status LEDs are controlled by NetROM's software. Figure 2-1 shows the front panel view of NetROM's LEDs.

The labeled LEDs in Figure 2-1 are the network LEDs and represent different network states.

RX	Indicates when NetROM is receiving frames.
TX	Indicates when NetROM is transmitting frames.
LINK	Represents twisted pair MAU link status.
POLARITY	Indicates reversed polarity on receives.

The numbered LEDs are the target status LEDs and can be mapped to the status signals on NetROM. By default LED 0 is used as a "heartbeat" LED, which indicates that NetROM is alive and gives some indication of the load on the system. The commands to map LEDs to status signals are described in Chapter 4. The single red LED to the left indicates that power is on.

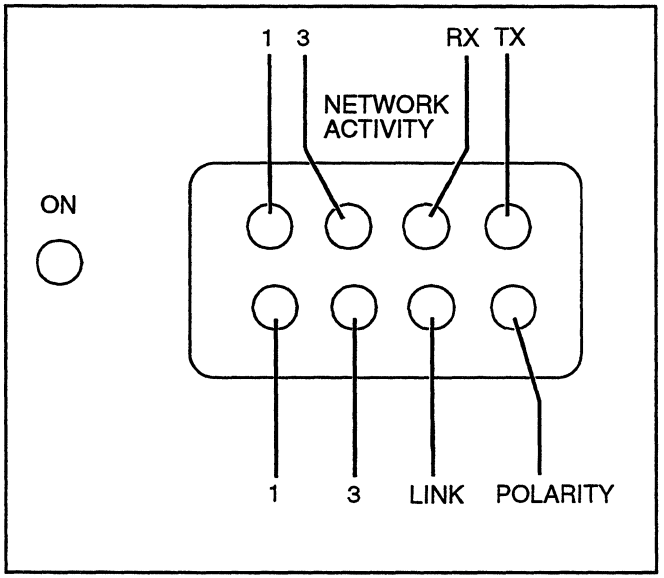


Figure 2-1 NetROM Back-Panel LEDs

Chapter 3
Installation

This chapter provides instructions for installing NetROM. The process is broken into hardware setup and software setup. The software section is further broken down into NetROM and host setup.

For detailed installation instructions, see the *NetROM Installation Notes* in the tabbed section at the end of this manual.

Collecting Equipment

Caution 

NetROM contains components that are subject to damage from electrostatic discharge. Whenever you are using, handling, or transporting the hardware, or connecting to or disconnecting from a target system, always use proper anti-static protection measures, including using static-free bench pads and grounded wrist straps.

Before starting the installation verify that you have the following equipment:

- An AC power cord and transformer provided with your NetROM.
- A set of ROM emulation cables. The standard NetROM is shipped with enough ROM emulation cables to support up to four Dual In-line Package (DIP) EPROMs. Other types of EPROMs may require additional cables (Please contact Applied Microsystems if you have special cable needs.)

- A means of connecting to the Ethernet. Ethernet connection may be accomplished in one of three ways:

  1. Via a transceiver,

  2. Via a twisted pair MAU,

  3. Via a RJ-45 twisted pair connector.

If you do not have the above mentioned items please obtain them before proceeding.

You may want to obtain a "dumb" terminal such as a VT100 to serve as a NetROM console while you troubleshoot your NetROM configuration. You may also want to check that equipment pictured in Figure 3-1 was shipped with NetROM. If not, please contact Applied Microsystems.

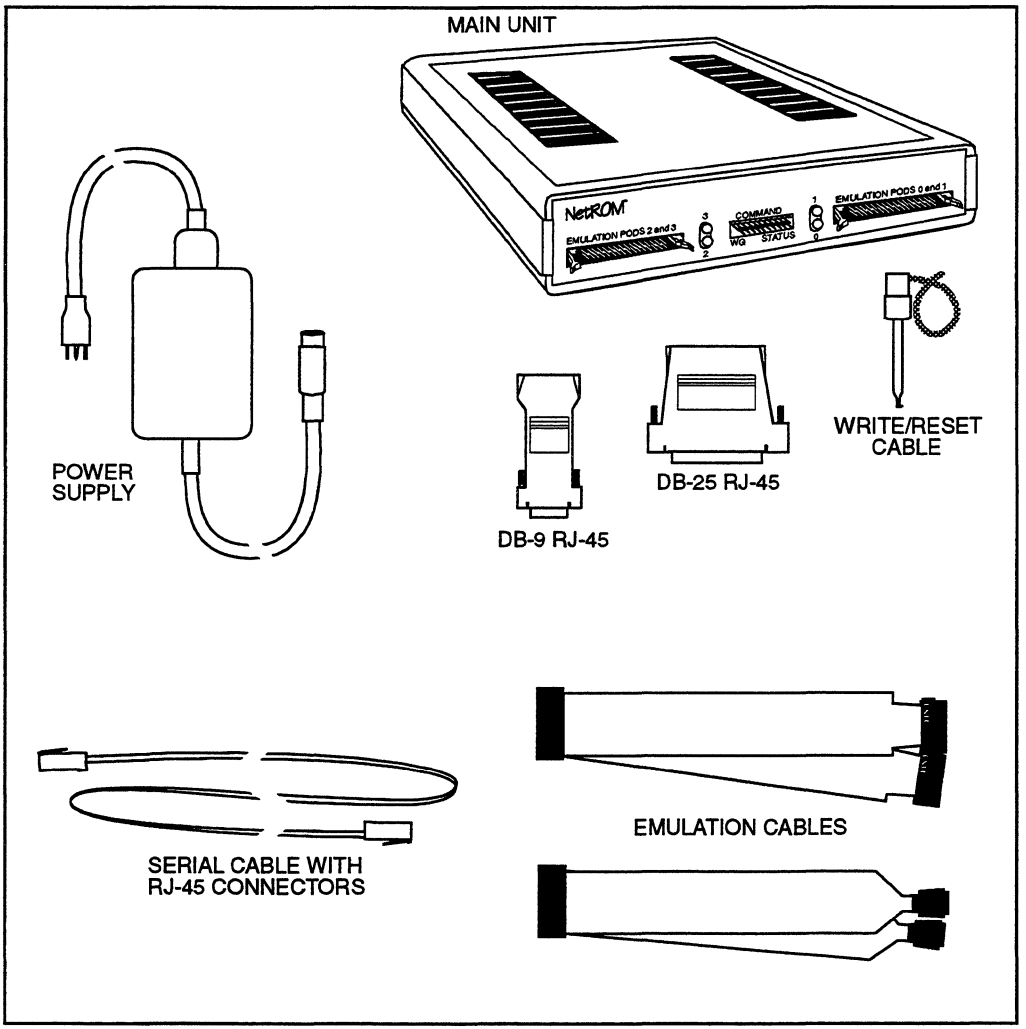


Figure 3-1 NetROM and Accessories

Hardware Setup

This section explains how to install external connections to the various connectors on NetROM. Figure 3-2 illustrates the front and rear panels of the NetROM, and the call-out numbers in the figure are referenced in the following sub-sections.

Caution

NetROM contains components that are subject to damage from electrostatic discharge. Whenever you are using, handling, or transporting the hardware, or connecting to or disconnecting from a target system, always use proper anti-static protection measures, including using static-free bench pads and grounded wrist straps.

Connecting AC Power Cord

An AC power cord with a transformer assembly is supplied with NetROM.

Make certain the power switch on NetROM is turned off, then attach the power cord to the connector labeled "power" on the rear panel of NetROM. Plug the other end of the power cord into a grounded AC wall outlet.

Caution

To avoid damaging NetROM use only the AC power cord and transformer supplied with your NetROM.

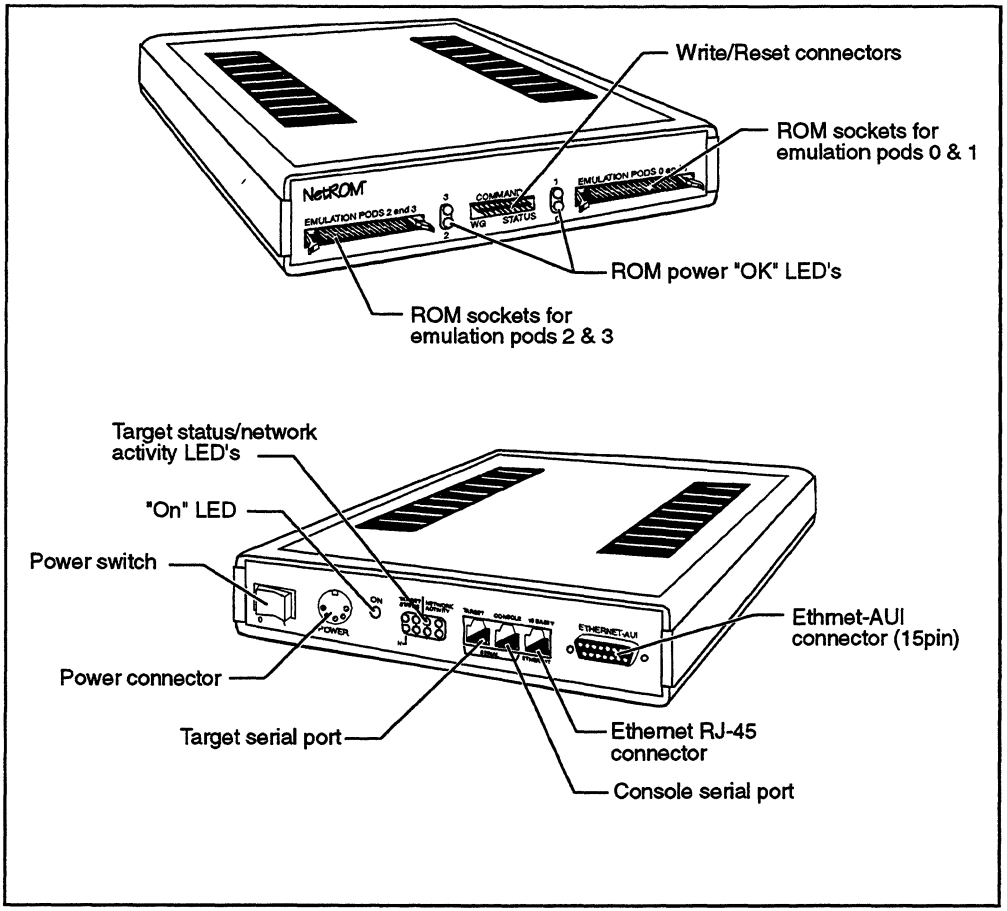


Figure 3-2 Connecting Hardware

Connecting to Ethernet

NetROM may be connected to the Ethernet via one of two means. Figure 3-2 shows the location of the connectors on the back of the NetROM, and Figure 3-3 shows the two connectors in detail. Note that no switches or jumpers need be set on NetROM when changing the type of Ethernet connection. NetROM automatically configures for whichever network

connection is plugged in. If *both* connectors are plugged in, NetROM will use the 10 Base-T interface and ignore the AUI transceiver.

Using an Ethernet Transceiver

An Ethernet transceiver is required when connecting NetROM to either thin or thick Ethernet cable. When using this type of Ethernet connection the transceiver is connected via an Attachment Unit Interface (AUI) cable to the 15-pin AUI connector on the rear panel of NetROM (Figure 3-3). Other connections for the transceiver should be performed according to the instructions supplied by the transceiver manufacturer.

Using an RJ-45 Connector

NetROM may also be connected to 10 Base-T networks via an RJ-45 connector (Figure 3-3).

Caution 

There are three RJ-45 connectors on the back panel of NetROM. Do not connect the 10 Base-T network to either connector labeled serial. The 10 Base-T connector should ONLY be connected to the connector labeled Ethernet. Also, do not connect any RS-232 cables to the connector labeled Ethernet. The Ethernet connector supplies a 12 Volt signal and may damage your RS-232 equipment.

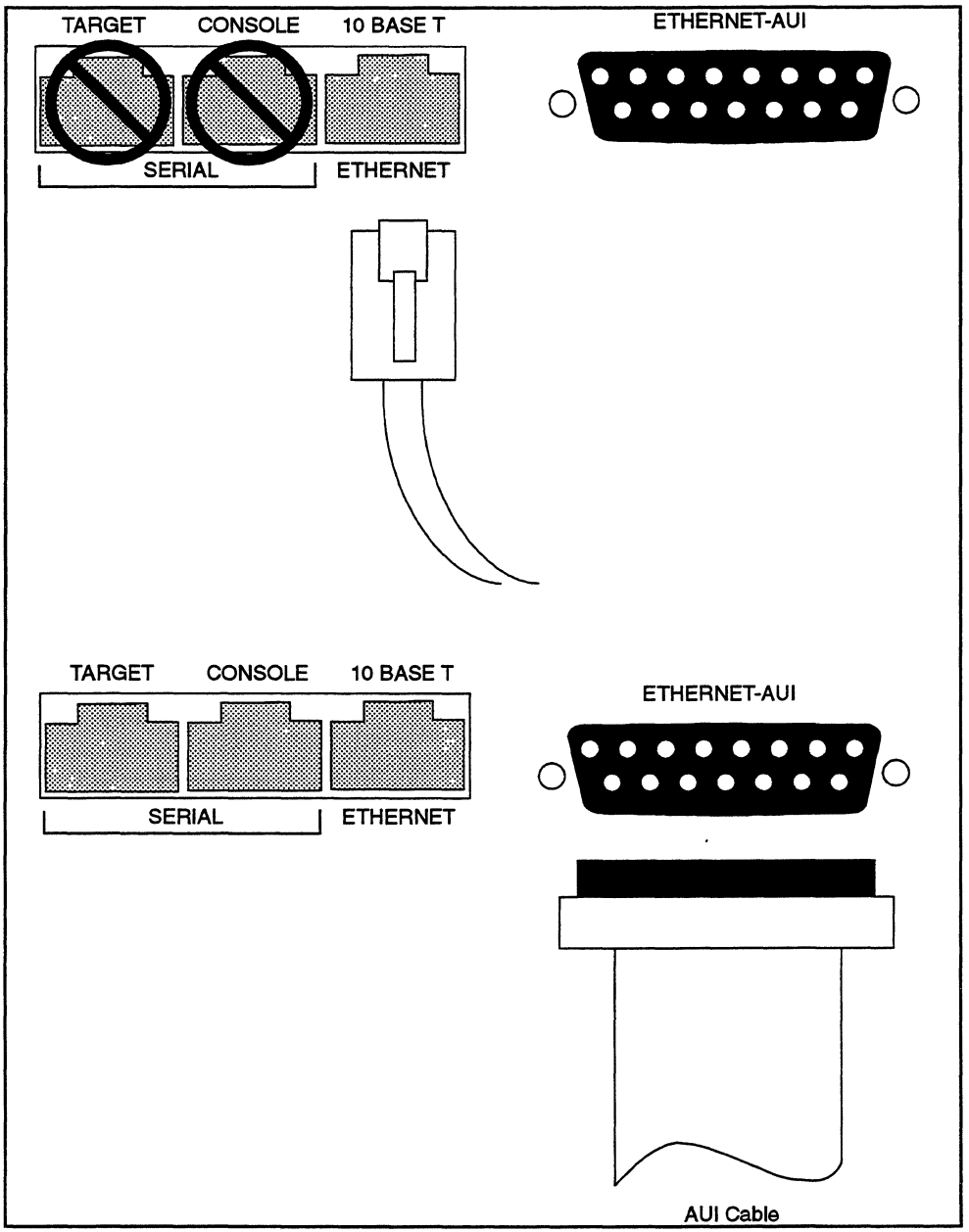


Figure 3-3 Connecting Ethernet

Connecting ROM Emulation Cables

There are multiple versions of the ROM emulation cables available for NetROM. ROM emulation cables vary depending on the format of the ROM package located on the target system. The cables all connect to NetROM through 80-pin female emulation pod connectors. Emulation pods 0 and 1 are connected to the NetROM socket shown in Figure 3-2; emulation pods 2 and 3 are connected the NetROM socket shown in Figure 3-2. Refer to "podorder" on page 4-117.

Connecting ROM emulation cables to the target system requires an understanding of the target system's word size and byte ordering and their relationship to NetROM's four (4) emulation pods. The NetROM software has a default mapping of emulation pods to target byte ordering. This mapping may be overridden using the "podorder" environment variable discussed in Chapter 4. However for this discussion we shall use the default mapping. In addition, the examples given are for targets which use big-endian byte ordering; little-endian targets will be discussed at the end for completeness.

NetROM users should understand the terms parallel and serial emulation. If the target system has a multiple-byte word size and more than one ROM is used to create this word, then you will be using parallel emulation. Parallel emulation thus refers to the operation of two or more NetROM emulation pods configured in parallel to provide a single word to the target system. If a target uses additional ROMs to provide more memory space than is supplied in a single ROM then serial emulation is used. For example, suppose a target has a word size of 8 bits and uses two 27c020 ROMs. This gives a memory space of 512 Kilobytes, where bytes 0 through 256K-1 are supplied by one ROM and bytes 256K through 512K-1 are supplied by the second ROM. This is an example of serial emulation. with NetROM it possible to use serial emulation, parallel emulation, and a combination of both. The following table shows the possible combinations of emulation based on the number of ROMs on a target system.

Table 3-1 Series and Parallel ROM Emulation

	Number of ROMs Emulated			
Word size	**1**	**2**	**3**	**4**
8 bit	serial	serial	serial	serial
16 bit	see note	parallel	see note	serial & parallel
32 bit	see note	see note	see note	parallel

Note

These configurations are possible but beyond the scope of this discussion.

Once you have determined the type of emulation you will be using (serial, parallel or both), you will need to determine which NetROM emulation pod to plug into the ROM sockets on your target system. As mentioned above this discussion will detail how to choose the proper configuration based on NetROM's default podorder settings.

For systems with an 8-bit word size there are four possible configurations, namely using 1, 2, 3 or 4 ROMs. Figure 3-2 shows mapping between NetROM emulation pods and the target's ROM addresses assuming a 256 Kilobyte (27c020) ROM. Also included are the data bits supplied by each emulation pod.

Table 3-2 Emulation of an Eight-bit Word Size

	Number of ROMs Emulated				
Pod	**1**	**2**	**3**	**4**	**Data Bits**
0	bytes 0 through (256K-1) 0x00000 to 0x3ffff	bytes 0 through (256K-1) 0x00000 to 0x3ffff	bytes 0 through (256K-1) 0x00000 to 0x3ffff	bytes 0 through (256K-1) 0x00000 to 0x3ffff	D0-D7
1	N/A	bytes 256K through (512K-1) 0x40000 to 0x7ffff	bytes 256K through (512K-1) 0x40000 to 0x7ffff	bytes 256K through (512K-1) 0x40000 to 0x7ffff	D0-D7
2	N/A	N/A	bytes 512K through (768K-1) 0x80000 to 0xbffff	bytes 512K through (768K-1) 0x80000 to 0xbffff	D0-D7
3	N/A	N/A	N/A	bytes 768K through (1M-1) 0xc0000 to 0xfffff	D0-D7

For systems with a 16-bit word size, there are two possible configurations: namely using 2 or 4 ROMs. Table 3-3 shows the mapping between NetROM emulation pods and the target's ROM addresses assuming a 256 K byte (27c020) ROM. Also included are the data bits supplied by each emulation pod.

Table 3-3 Emulation of a 16-bit Word Size

Pod	Number of ROMs Emulated 1	2	3	4	Data Bits
0	N/A	even bytes 0 through (512K-1) 0x00000 to 0x7ffff	N/A	even bytes 0 through (512K-1) 0x00000 to 0x7ffff	D8-D15
1	N/A	odd bytes 0 through (512K-1) 0x00000 to 0x7ffff	N/A	odd bytes 0 through (512K-1) 0x00000 to 0x7ffff	D0-D7
2	N/A	N/A	N/A	even bytes 512K through (1M-1) 0x80000 to 0xfffff	D8-D15
3	N/A	N/A	N/A	odd bytes 512K through (1M-1) 0x80000 to 0xfffff	D0-D7

For systems with a 32-bit word size there is one possible configuration: using four ROMs. Table 3-4 shows mapping between NetROM emulation pods and the target's ROM addresses assuming a 256K byte (27c020) ROM. Also included are the data bits supplied by each emulation pod.

Table 3-4 Emulation of a 32-bit Word Size

	Number of ROMs Emulated				
Pod	**1**	**2**	**3**	**4**	**Data Bits**
0	N/A	N/A	N/A	byte zero, 0 through (1M-1), 0x00000 through 0xfffff	D24-D31
1	N/A	N/A	N/A	byte one, 0 through (1M-1), 0x00000 through 0xfffff	D16-D23
2	N/A	N/A	N/A	byte two, 0 through (1M-1), 0x00000 through 0xfffff	D8-D15
3	N/A	N/A	N/A	byte three, 0 through (1M-1), 0x00000 through 0xfffff	D0-D7

When connecting the ROM emulation cables, it is helpful to understand the byte ordering on the target system. The tables above have shown the mapping for big-endian systems. The next figure shows the byte ordering for big- and little-endian systems for 16- and 32-bit word sizes. Using this figure and the previous tables, it should be possible to determine which NetROM emulation pod to plug into the target system's ROM sockets.

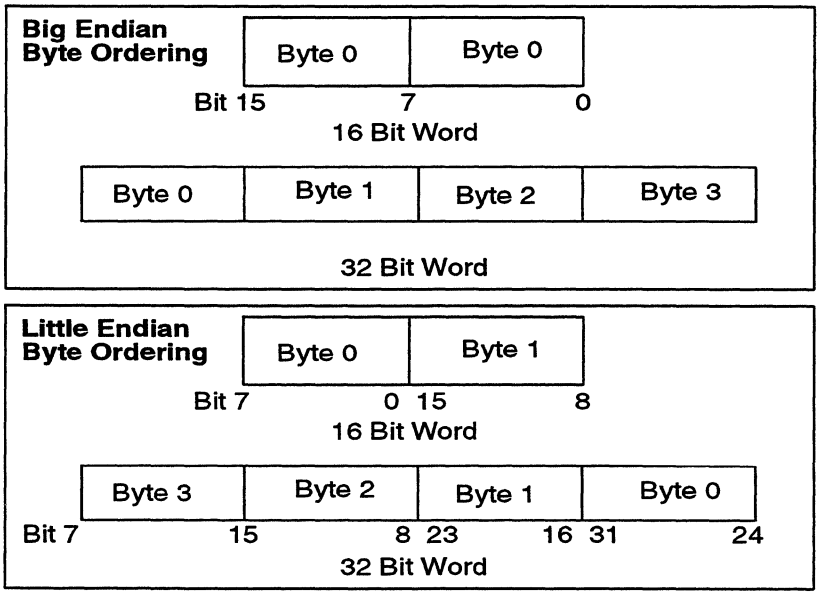


Figure 3-4 Big-endian/Little-endian Byte Ordering

Connecting DIP Style Cables

DIP-style cables are used to connect NetROM to targets that contain DIP ROM packages. Available in 28-, 32-, and 40-pin versions, these cables plug directly into the ROM sockets on the target system. Before connecting the emulation cables to the target system, be certain to turn off power to the target system. Remove any ROMs currently in the ROM sockets on the target. *Be certain to align pin 1 on the ROM emulation cable DIP connector with pin 1 on the target DIP socket.* The ROM power "OK" LEDs (see Figure 3-2) will light when the cables are plugged in correctly and the target system is powered on.

Caution

Plugging in ROM cables improperly may damage the NetROM.

Connecting PLCC Style Cables

PLCC style cables are used to connect NetROM to targets which use PLCC ROM packages. These cables plug directly into the ROM sockets on the target system. Before connecting the emulation cables to the target system, be certain to turn off power to the target system. Remove any ROMs currently in the target sockets. *Be certain to align pin 1 on the emulation cable to pin 1 on the target PLCC socket*; one corner of the emulation cable's PLCC plug is cut off to indicate the location of pin 1.The power "OK" LEDs (Figure 3-2) will light when the cables are plugged in correctly and the target system is powered on.

Connecting NetROM Console

The NetROM console connection allows the NetROM user to communicate with the NetROM executive via a serial device such as a terminal. Figure 3-2 shows the location of the console port on the rear of the NetROM, and Figure 3-5 is a view of the NetROM console socket. The pinouts for the NetROM console connector are shown in Appendix A, and Appendix E gives the default configuration of the Console Serial Port. The serial connection should be made using an RJ-45 connector. The 9- and 25-pin connectors supplied with NetROM are DTE, not DCE. The DTR signal is always "true," and that the DSR signal is ignored.

Connecting Target Serial Port

The NetROM target serial port allows the NetROM user to access the target system's serial port. Figure 3-2 shows the location of the target serial port on the rear of the NetROM, and Figure 3-5 is an exploded view of the target serial socket. The pinouts for the NetROM serial port are shown in Appendix A, and Appendix E gives the default configuration of the Target Serial Port.

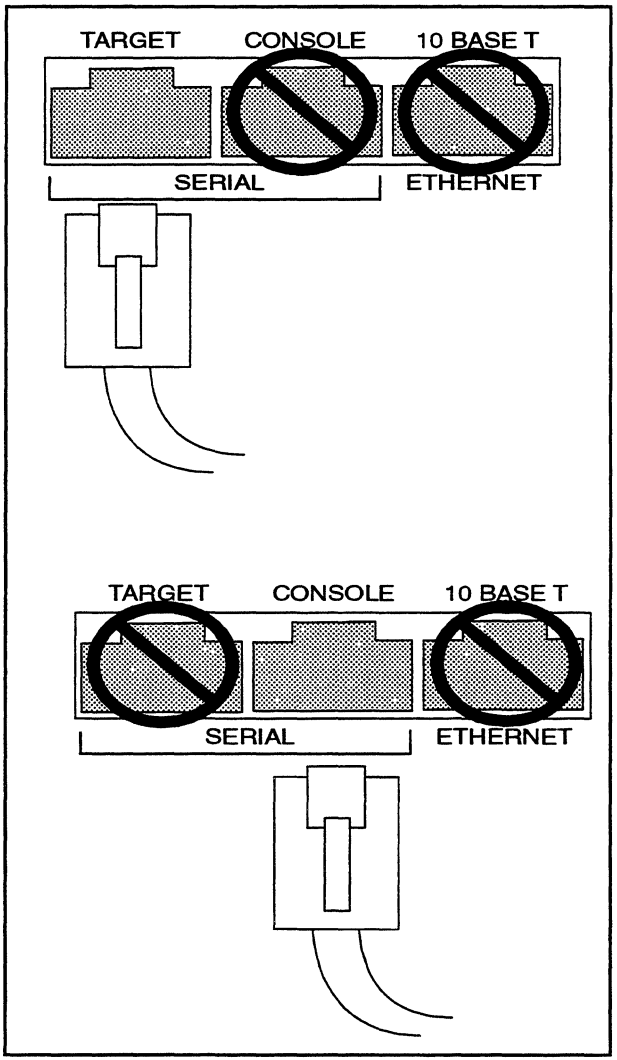


Figure 3-5 Connecting NetROM Serial Ports

The serial connection should be made using an RJ-45 connector. The 9- and 25-pin connectors supplied with NetROM are DTE, not DCE. The DTR signal is always "true," and that the DSR signal is ignored.

Connecting Trace Cables

Trace cables are not yet provided with NetROM.

Connecting the Write Signal

The Write signal connection allows target systems to write directly to their ROM space. The data written by the target system is deposited at the specified address in NetROM's emulation memory. Writing to your ROM address space requires a a Write signal. There are three methods to do this:

- Have the target software monitor request NetROM to write emulation memory.
- Allow your target hardware to signal a Write access on the PGM pin of the ROM socket. This method is normally the case if NetROM is plugged into sockets designed for FLASH ROM. An additional requirement is that your NetROM cable must support writing. To see if your cable type supports writing, refer to Table 4-11 in "romtype" on page 4-120.
- Connect a Write signal somewhere else on the target board. This method also requires the cable to support writing. To see if it does, refer to Table 4-10 in "romtype" on page 4-120.
- If you need to connect the Write signal, connect a jumper cable from the target system's write strobe to the Write pin on the front panel of NetROM (Figure 3-2 and Figure 3-6). The write signal is "active low" and is expected to occur in conjunction with a normal write cycle.

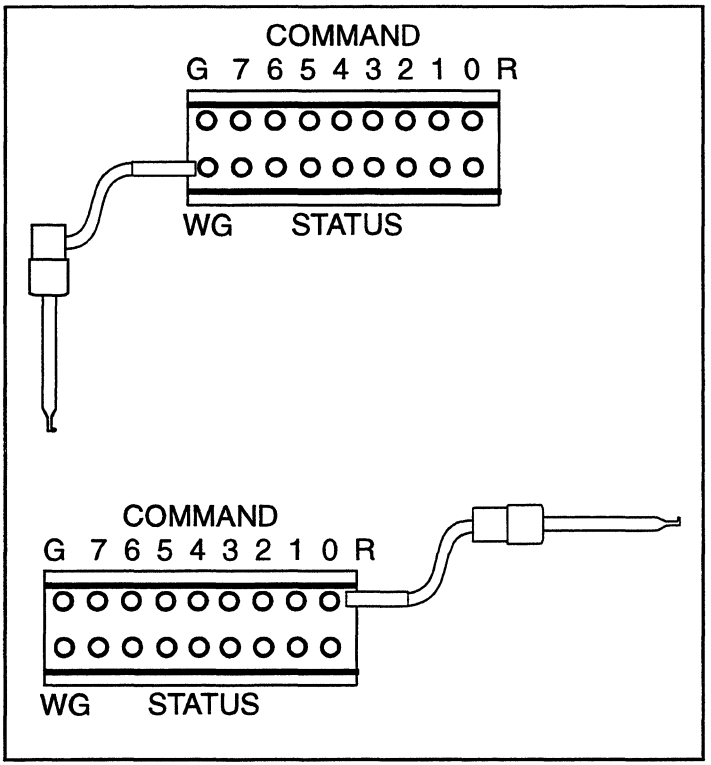


Figure 3-6 Connecting Wire/Reset Cables

After the cable is connected, software running on the target may treat the ROM space as writable memory. This allows easy insertion of breakpoints and/or patching of code by the target.

Connecting the RESET Signal

The Reset signal is output by NetROM to reset the target system. This allows the NetROM user to reset their target system remotely. A jumper cable should be connected from the target's Reset signal to the Reset pin on the front panel of NetROM (see Figure 3-2).

The Reset signal is "active low"; when it is asserted, it connects to ground. The Reset signal should be connected to "open-collector" traces, or to traces which drive a small amount of current. An example configuration is shown in Figure 3-7.

Caution

If NetROM's Reset signal is connected to a high-current trace on the target system, assertion of the signal may cause a short circuit! Consult "NetROM LEDs" on page 2-13 for more information on NetROM command signals.

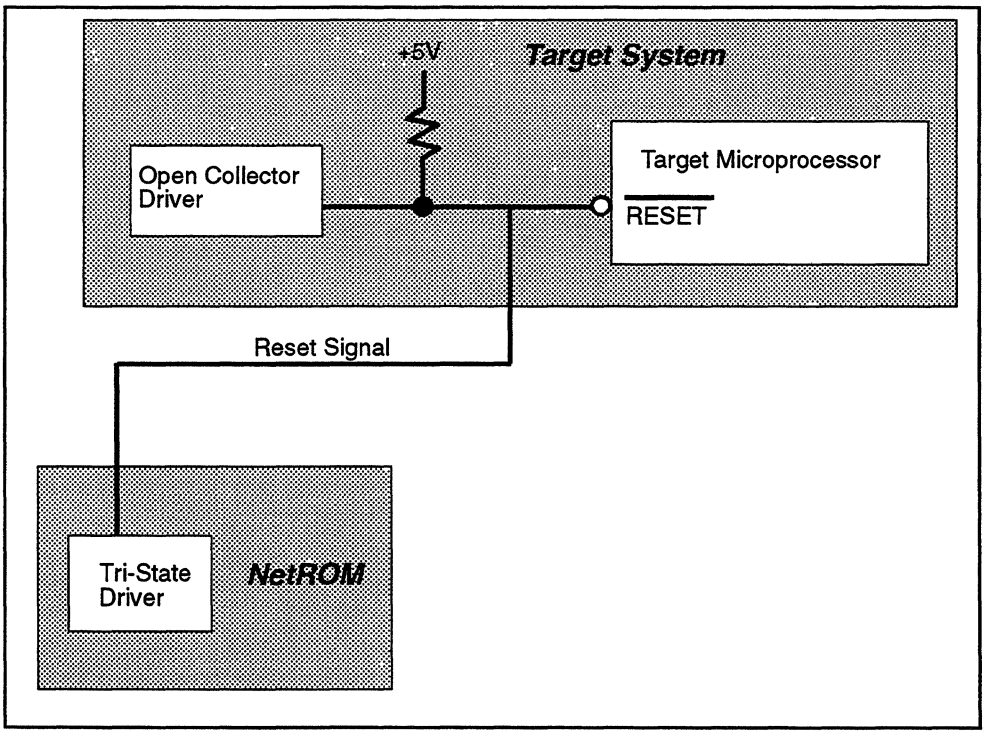


Figure 3-7 Connecting NetROM's Reset Signal

Software Setup

NetROM's hardware installation consists mainly of plugging cables and power cords into the appropriate connectors. Software installation is somewhat more involved, since NetROM is a full-fledged multitasking network host, capable of supporting multiple simultaneous terminal sessions and other types of network connections. The initial effort required to install NetROM on a network is comparable to that required when installing a new workstation. However, once NetROM is a functioning member of your network, using it is quite simple, even if you physically move the unit or use it on different target systems.

To use NetROM most effectively, it helps to have address resolution and file serving software running on a host system. NetROM's software setup essentially consists of configuring one or more host systems to provide these two basic services. At power-up, NetROM knows its Ethernet address, but not its IP address. While it is possible to configure NetROM's network address directly, using NetROM's serial console, it is much simpler to have NetROM resolve its network address automatically. File server support, in the form of a TFTP server, should be present on your network. This is how NetROM will download images into emulation memory. Once address resolution and file server software is running in your development environment, NetROM can be used anywhere there is an Ethernet connection. Serial lines and "dumb" terminals will no longer be necessary to debug your target system.

This section describes, in general terms, the network protocols which NetROM uses to perform address resolution and file download functions. Consult the *Installation Notes* for tips on how to configure host-side servers for your development environment. Once servers are configured, it is only necessary to create a startup batch file for your particular project. Startup batch files are discussed in the last part of this section.

Address Resolution

As mentioned before, at power-up a NetROM unit does not know its own IP address, but it does know its hardware (Ethernet) address. Determining one's own network address is a common problem for diskless computers, such as diskless workstations. Such computers require a mechanism to determine their IP address, knowing only their hardware address. Two software protocols are commonly used for this purpose; BOOTP and RARP. When NetROM powers up, it sends out simultaneous RARP (Reverse Address Resolution Protocol) and BOOTP requests. These requests are broadcasts, so all systems on the Ethernet "hear" them. BOOTP (or RARP) "servers" listen for these requests, look up the requestor's IP address based on the hardware address in the request frame, and send a response. If NetROM receives a response to either its BOOTP or RARP request, it uses the information in the response to configure its IP address. If no response is received, it will re-broadcast the requests 10 times and finally give up. The NetROM unit should then be either power cycled or configured from the NetROM console.

BOOTP and RARP are simple protocols; often, the most complicated part of configuring a host computer as a "server" is setting up configuration files and making sure the server daemon is running! Once the server is responding to NetROM's requests, NetROM should have no problem determining its IP address, regardless of how often it is reset, power cycled, or moved. Consult the *Installation Notes* addendum for details on configuration file setup.

If both BOOTP and RARP are available in your development environment, the choice of which to use should be based on convenience. If NetROM receives both a BOOTP and a RARP response simultaneously, it will use the BOOTP response to configure its address.

BOOTP Address Resolution

BOOTP, like TFTP, is a protocol which uses the connectionless UDP transport protocol. There are two "flavors" of BOOTP servers, the "CMU" and the "Stanford" versions. These differ

only slightly; the CMU version allows the server to specify the client's IP netmask and broadcast address, as well as its IP address. Both versions provide a TFTP server address and a download filename to request from the server. NetROM understands BOOTP responses from either of the server types. However, the configuration file formats for the two types of servers are completely different, even though the configuration files have the same names. This often results in great confusion among people who are trying to set up BOOTP on their Ethernet, especially since errors in the configuration file cause the BOOTP server to quietly ignore requests!

If a configuration file is not explicitly specified in a BOOTP response, NetROM will not try to download one.

RARP Address Resolution

RARP is another way to establish a mapping between an Ethernet address and an IP address. Unlike BOOTP, RARP does not use UDP as a transport. In fact, it is a variant of ARP, the Address Resolution Protocol in common use throughout TCP/IP networks, which does not even use IP as a network protocol. Also unlike BOOTP, the RARP protocol does not provide a mechanism to specify a download file. However, there is a convention which allows the RARP client to determine what configuration file it should download, as well as determine its IP address.

NetROM will map RARP responses to startup file names as follows. The RARP response consists of an IP address, which is a 32-bit value. NetROM converts the individual numbers in its dotted-decimal Internet address to two-digit hex numbers. For example, given the address 192.0.0.210, 192 becomes 0xC0, 0 remains 0x00, and 210 becomes 0xD2. NetROM will then concatenate these numbers to produce the filename (in uppercase letters) *C00000D2*. NetROM will then attempt to download one of the following startup files: "C00000D2," then "/tftpboot/C00000D2," and finally "tftpboot/C00000D2." After the first successful download, it will proceed with its boot sequence and execute the commands in the startup file. It will *not* attempt to download other startup files.

Since the RARP protocol cannot specify a TFTP server, NetROM assumes that the TFTP server for the configuration file is the same as that which responded to the RARP request. See "File Server Support" (next section) for further information on TFTP.

If this configuration file does not exist, or if there is not a TFTP server running on the RARP server host, NetROM will display appropriate error messages on its serial console and use its default configuration values (see Appendix E).

File Server Support

TFTP is a standard file transfer protocol often used to provide boot images to standalone devices. The TFTP server can be run on any number of nodes on a network. By default, it looks for files in the directory */tftpboot;* that is, if the server receives a request for file *startup.bat* it will respond by sending the file */tftpboot/startup.bat*.

The server can be run in either of two modes: "secure" mode or "normal" mode. These two modes differ primarily in the way they handle root-specific filenames in requests. Root-specific requests are those which specify directory location starting at the root of the directory tree. For example, the client may request the file */startup.bat*. Servicing this kind of request could lead to a security risk; the client could request any file on the server system; and since TFTP performs no authentication, the client would get the file. Secure TFTP was developed to get around this problem. Secure TFTP servers respond to root-specific requests as if the directory */tftpboot* were the root directory. Secure servers cannot be circumvented using symbolic links or similar tricks.

The problem with secure servers is that they prevent legitimate clients from downloading files outside of the */tftpboot* directory subtree. For example, NetROM may want to download a file from an engineer's development directory, but secure-mode TFTP may make this impossible. There are at

least two ways of dealing with this problem; either do not use secure TFTP, or copy images into the */tftpboot* directory subtree after each modification.

As with BOOTP and RARP servers, it is often harder make sure that your TFTP server is running and getting files from where you think it is, than it is to request a file from NetROM. Consult the *Installation Notes* addendum for more information on installing TFTP servers.

NetROM Startup Files

NetROM allows the user to specify a startup batch file. Batch files are sequences of NetROM command-line commands delineated by begin/end statements. Chapter 4 provides details on batch files and how to use them. NetROM's address resolution mechanism at boot time determines what startup file it attempts to run, and which server it expects to provide the file. BOOTP responses explicitly name a startup file and a TFTP server, so when BOOTP is used to configure NetROM at boot time, it is easy to specify the startup file. When RARP is used as the address resolution protocol, NetROM uses its IP address to construct the name of the file.

An example startup file is the following:

```
begin
     setenv wordsize 16
     setenv romtype 27c020
end
```

This file tells NetROM to organize its emulation pods as 16-bit words, emulating 256 Kilobyte ROMs. The commands in the file are executed in order, just as if the NetROM user had typed them in at the keyboard. The file must be a "pure" text file; the editor you use cannot use unprintable formatting characters.

If no startup file is specified with BOOTP, NetROM will not attempt to download one. However, NetROM will always try to download the RARP configuration file (if its address is being configured by RARP). In this case, TFTP will inform it that the

file does not exist and no harm will be done. If TFTP is not running on the RARP server, NetROM will abort its download effort.

The output from the NetROM commands in the startup file will go to the NetROM console serial port. Unless there is a terminal connected to this port, errors in the startup file may not be noticed. It is a good idea to test changes to the startup file by running it as a batch file from the command line; this allows you to debug your startup file without having to connect a "dumb" terminal to the NetROM console.

Chapter 4
User Interface

This chapter describes the interface which NetROM presents to human users. Users will generally interact with NetROM via the NetROM console serial port, TELNET sessions, or direct TCP connection to the NetROM console port. This chapter will first describe command line processing performed by NetROM, then the actual commands themselves.

NetROM Command Line Processing

NetROM accepts any number of single-line commands. If a command is wider than the terminal on which it is entered (which it might be in a TELNET session), the command will wrap to the next line. There is no restriction on the length of command line arguments. However, the maximum length of the command line is 128 bytes, and the maximum number of command line arguments is 16. There are five major facets to NetROM's command line processing. These are processes, terminal control characters, environment variables, history substitution, and batch processing.

Processes

NetROM uses a multitasking operating system to provide services to the user. Each task running on NetROM is called a "process." Processes allow NetROM to divide responsibility for user services. Each terminal session, for example, is a separate process. Processes have both a name and a process identifier, or "pid," to identify them. More than one process may have the same name, so pids are used in NetROM's process control commands. Current status for all NetROM processes may be listed using the *ps* command, described below. Processes are generally *sleeping*, *ready*, *running*, or *yielding*, but there are a

few other, generally transient, states which may be displayed in the listing.

Each process has a controlling terminal and is capable of reading commands from it and writing status to it. Generally a single process is in charge of the controlling terminal and, if it spawns child processes, controls them directly. It is also possible to control processes using *signals*, which can be sent from other controlling terminals. Table 4-1 lists signals currently supported by NetROM's operating system.

In most cases, signals sent to processes will cause them to be killed, so this is not a good idea unless a process is *known* to be hung. Users will use the SIGINT signal to kill child processes of the issuing terminal. If the process ignores this signal, the SIGKILL signal will probably work. The SIGKILL signal cannot be ignored, but the receiving process may not be able to clean up its state before exiting, so SIGINT should be used in preference to SIGKILL. Finally, the SIGHUP signal may be used to restart certain "server daemons" running on NetROM, such as *snmpd*. However, using SIGHUP in this way is not currently implemented.

Other signals are used internally by the NetROM operating system and should not be sent by NetROM users.

Table 4-1 NetROM Signal Summary

Number	Name	Meaning
1	SIGHUP	The controlling terminal for a process has been terminated. For example, a TELNET session ended after spawning a *ping* process.
2	SIGINT	Interrupts (kills) another process. This is the standard way to terminate processes asynchronously, and is generated by the "intr" character on the controlling terminal of a process.
3	SIGKILL	This is a more fatal way of killing a process; it cannot be caught, blocked, or ignored. It should be used with great care.
4	SIGALRM	This signal is used internally by NetROM to indicate a timer event.
5	SIGPIPE	This signal is used internally by NetROM to indicate that a write attempt occurred on a closed socket.
6	SIGABRT	This signal is reserved, and causes the NetROM operating system to hang.

Terminal Control Characters

NetROM considers interactive "console" sessions to be running from terminals. Terminals may be attached to NetROM through the NetROM console serial port, TELNET connections, or direct TCP connections on the NetROM Command Port. Each interactive NetROM session has several control characters associated with it. These characters are considered "special" by the command interpreter and are used for command-line editing and process control.

The "erase" character erases the character to the left of the cursor from the input stream, if there is one present. The "kill" character erases all characters in the input stream and starts a new line. The "werase" character erases the white-space-

delimited "word" to the left of the cursor if one is present. The "intr" character sends a SIGINT signal to all child processes of the current session. Child processes might be *ping* processes, file downloads, or any other process started as a result of a command on that terminal. NOTE that the "intr" character is treated specially during target console sessions: if the console path is serial and an "intr" is detected, NetROM will send a BREAK to the target. The "eof" (end-of-file) character is used to indicate that interactive input for a given command, such as a target console, is complete. The "eof" character will not terminate a terminal session. Table 4-2 summarizes the terminal control characters.

Control characters may be displayed or set using the *stty* command, described below. Control characters may also be set for all subsequent terminal sessions using the same command. This is useful for establishing default control characters at NetROM reset.

Table 4-2 Terminal Control Characters

Name	Value	Description
eof	^D	End of file.
erase	^H	Erase the character to the left of the cursor.
intr	^C	Send a SIGINT signal to all child processes. (See also "tgtcons" on page 4-28.)
kill	^U	Erase the input line.
werase	^W	Erase the "word" to the left of the cursor.

Environment Variables

Environment variables affect all terminal sessions running on NetROM. All environment variables are predefined; they are primarily concerned with configuring emulation and with establishing communications paths between the target and the host system. NetROM environment variables are summarized in Table 4-3. They are described in more detail under the *setenv* and *printenv* commands, below.

Table 4-3 NetROM Environment Variables

Variable Name	Description
"batchpath"	The default directory on NetROM's TFTP file server to search for batch files.
"consolepath"	A keyword (*serial, readwrite,* or *readaddr)* specifying the console communication path between NetROM and the target system.
"debugpath"	A keyword (*serial, readwrite,* or *readaddr)* specifying the debug communication path between NetROM and the target system.
"debugport"	Set the TCP/UDP port number for host-based debuggers.
"dprbase"	The base address in emulation pod 0 for mapping dualport RAM.
"filetype"	The file format expected of the download file.
"fillpattern"	User-specified byte pattern to fill emulation memory.
"groupaddr"	The target's start address of the default podgroup.
"groupwrite"	A keyword (*readonly* or *readwrite*) indicating whether to enable target writes to emulation memory.
"host"	The IP address of the TFTPfile server used for image and batch downloads.
"loadfile"	The default file to download into the default pod group.

Table 4-3 NetROM Environment Variables (Continued)

Variable Name	Description
"loadpath"	The default path for downloading "loadfile."
"podgroup"	The group number of the default pod group.
"podorder"	The pod-to-byte mapping of emulation pods in the default pod group.
"romcount"	The number of bytes participating in emulation as part of the default pod group.
"romtype"	The type of ROM being emulated by the default pod group.
"verify"	A keyword (*on or off*) specifying whether downloads are to be verified.
"wordsize"	The size in bits of the ROM word being emulated by the default pod group.
"writemode"	Configures emulation memory to emulate flash ROM or static RAM.

History Substitution

Each NetROM terminal session remembers commands it has been given. Remembered commands are said to be in the NetROM "history buffer." Currently the history buffer for each session is 16 commands deep. Commands in the history buffer may be repeated and/or edited in a style similar to the UNIX *csh* command interpreter.

➤ **To modify and repeat the most recent command**

^aaa^bbb Replaces the string aaa with string bbb in the most recent command.

➤ **To repeat a recent command**

!!	Repeats the most recent command.
!*nn*	Repeats command number nn.
!*aaa*	Repeats the command beginning with the string aaa.
!?*aaa*	Repeats the command containing the string aaa.

➤ **To add a string to the end of a previous command and repeat it**

!!*aaa*	Adds string aaa and repeat the most recent command.
!nn aaa	Adds string aaa and repeat command number nn.
!*aaa bbb*	Adds string bbb and repeat the command beginning with the string aaa.
!?*aaa bbb*	Adds string bbb and repeat the command containing the string aaa.

Batch Processing

NetROM allows users to create "batch files" on the host system. Batch files are simply multiple NetROM commands collected into a file. The file should be delimited by *begin* and *end* statements, and may have comments (identified by a pound sign (#) as the first character on the line). Note, however, that comments must be on lines by themselves. *Batch files should consist only of ASCII text, and should not be greater than 2048 bytes in size.*

Batch files can be invoked on the command line using the *batch* command; see Table 4-4. When it processes a *batch* command, NetROM downloads the file from its TFTP server (given by the

"host" environment variable) and executed one line at a time. The batch file's path on the server can be given explicitly on the command line or inferred from the "batchpath" environment variable. An example batch file is shown below:

```
begin
# download a new image and reset the target
newimage
tgtreset
end
```

If this file were called "new" and were located in the batchpath directory, executing the command

```
NetROM> batch new
```

would execute first the *newimage* command and then the tgtreset command, in order. The comment is parsed and ignored. Commands executed within the batch file will be entered into NetROM's history buffer. Batch files may call other batch files.

NetROM Commands

Commands can be issued to NetROM through a TELNET connection, NetROM's Console Serial Port, or via a network connection to the NetROM Control Port. This section describes all of the commands available to the NetROM user interface. Commands are grouped in functional sections, such as network interface control, target download and control, and process control. Table 4-4 summarizes commands alphabetically and gives their types. Composite commands, such as *set*, *di*, *setenv*, and *printenv* are treated in separate sections

Table 4-4 NetROM Command Summary

Command	Type	Description
alias	Miscellaneous	Creates or deletes command "nicknames."
arp	Network interface	Displays or modifies the contents of the NetROM Address Resolution Table.
batch	Miscellaneous	Downloads and executes a batch file containing NetROM commands.
di	Set and display	Displays various "generic" NetROM state variables, statistics, and target statistics information.
fill	Target interface	Fills emulation memory with a known pattern.
help	Miscellaneous	Accesses NetROM on-line help facility.
history	Miscellaneous	Displays the contents of the history buffer for the current NetROM session.
ifconfig	Network interface	Displays or configures a network interface.
kill	Process control	Sends a signal from one process running on NetROM to any other process.
ledmap	Miscellaneous	Maps NetROM's target status signals to LEDs on the back panel.
loadmodule	Miscellaneous	Loads the RAM-based optional software.
logout	Miscellaneous	Terminates a login session.
netstat	Network interface	Displays network statistics.
newimage	Target interface	Downloads a file into emulation memory.
ping	Network interface	Determines whether remote hosts are up and accessible.

Table 4-4 NetROM Command Summary (Continued)

Command	Type	Description
printenv	Environment variable	Displays the current values of NetROM's environment variables.
ps	Process control	Displays the current status of processes running on NetROM.
reset	Miscellaneous	Resets NetROM hardware and software.
romset	ROM set	Manipulates large ROM address spaces or word sizes greater than 32 bits.
route	Network interface	Manipulates information in NetROM's IP routing table.
serialcons	Target interface	Creates a "console" on a non-target system using NetROM's target serial port.
set	Set and display	Sets or modifies various NetROM state variables.
setenv	Environment variable	Modifies the value of environment variables.
slip	Network interface	Attaches or detaches a serial line to the Serial Line IP interface.
stty	Miscellaneous	Displays or modifies characteristics of NetROM terminal sessions.
tgtcons	Target interface	Establishes a console session with the target system.
tgtreset	Target interface	Resets the target processor.

For example the description:

stty [*-d*] { *erase* | *kill* | *werase* | *intr* | *eof* } **setting**

describes the *stty* command, for which the '-d' argument is optional, but which requires one of the keywords *erase*, *kill*, *werase*, *intr*, or *eof* followed by an argument, **setting**. Since **setting** is not a keyword, it will presumably be described in the text of the command documentation.

Commands which have multiple formulations will have each version appear on a line by itself. For example, the description:

arp dump
arp del **host_address**
arp set **host_address hardware_address**

indicates that the *arp* command can be invoked in any of the three ways shown.

When describing P addresses, NetROM commands use standard Internet "dotted-decimal" notation. An example of such an address is "192.0.0.210"; this corresponds to the hexadecimal number 0xC00000D2, but is expressed with each octet (that is, byte) expressed as a decimal number separated from the next by a period.

NetROM uses a similar format to describe Ethernet hardware addresses. However, there are three important differences:

- Ethernet addresses are 6 octets long, not 4.
- Octets are separated by colons.
- Octets are expressed in hexadecimal.

For example, the address 0x0002F4000024 is expressed as "00:02:f4:00:00:24." NetROM is not case sensitive in address representation. We will refer to this as "colon-separated hexadecimal" format.

Network Interface Commands

NetROM has several commands which control its various network interfaces. These include *arp, ifconfig*, *netstat*, *ping*, *route*, and *slip*. Most of these commands are similar to UNIX commands of the same name. They are generally used by system administrators to configure NetROM for operation in particularly complex environments, or to verify that it is interacting with other network hosts in the expected way.

The function of some of these commands, such as *ifconfig*, are performed automatically during the address resolution phase of NetROM's boot sequence. Others, such as *route*, *arp*, or *slip*, can be added to the NetROM startup file. This also causes them to be invoked automatically at NetROM boot time.

arp

Used to display and/or modify the contents of NetROM's Address Resolution Table.

Synopsis

```
arp dump
arp del host_address
arp set host_address hardware_address
```

Description

When IP is run over Ethernet, hosts on the network must be able to determine the Ethernet address of hosts with a given IP address. This mapping is provided by ARP (Address Resolution Protocol). NetROM maintains an Address Resolution Table, or ARP table, which contains mapping information about hosts NetROM has "seen" on the network.

The *arp* command displays or modifies the contents of NetROM's ARP table. It can be used to dump the table, add new entries, or delete current entries. All host addresses must be in dotted-decimal notation, and hardware addresses are in colon-separated hexadecimal format.

ifconfig

Displays or modifies the address, netmask, broadcast address, or operating state of one of NetROM's interfaces.

Synopsis

```
ifconfig
ifconfig ifname [ ifaddress ] [ netmask maskval ] [
broadcast broadaddr ]
ifconfig ifname ifaddress destaddr
ifconfig ifname { up | down }
```

Description

The *ifconfig* command can be used to configure either of NetROM's two network interfaces: the Ethernet interface and the SLIP interface. The SLIP (Serial Link IP) interface runs through the NetROM console serial port, and can be used to connect NetROM to the host computer when an RS-232 connection is not desired. Generally, the NetROM console port is not used, or is used to connect to a "dumb" terminal. The **ifname** parameter which refers to the Ethernet interface is "le0," and the one referring to the SLIP interface is "sl0." In addition, NetROM has a "loopback interface", which does not connect to external hardware. This can be used to verify that NetROM's TCP/IP protocol stack is working properly by sending "ping" packets to NetROM's IP address, but most users can safely ignore it. This interface's ifname is "lo0."

The first formulation of the *ifconfig* command is used to display state information about all of NetROM's network interfaces. Information displayed will include IP address, netmask, broadcast address, and whether the interface is up or down. Input, output, and error statistics will also be displayed for each interface.

The second formulation is used to set IP parameters for a given interface. It is possible to set the IP address, netmask, or broadcast address, or more than one of these addresses, using this form of the command.

The third formulation is used to configure the point-to-point SLIP link. Since SLIP is not a broadcast protocol, IP needs to know the address of the host at the other end of the serial line.

The final formulation of the command is used to enable or disable network interfaces. This command should be used with care, since it is possible to disable the interface on which the command was issued!

Note

All addresses, **ifaddress**, **destaddr**, **maskval**, and **broadaddr**, should be given in dotted-decimal format. Although this command can be used to set NetROM's Ethernet address manually, it is simpler and probably more convenient to use RARP or BOOTP to perform address resolution when NetROM is reset.

---

netstat

Displays network statistics and routing information.

Synopsis

netstat [*tcp* | *udp* | *ip* | *icmp* | *routes*]

Description

The *netstat* command, when issued without arguments, displays information about NetROM's TCP and UDP "connections." This consists of the local and remote addresses of bound sockets, and for TCP, the current state of the connection. Usually the TCP state is either LISTEN or ESTABLISHED. Addresses are displayed in a special five-field dotted-decimal format. The first four fields are the standard IP address, and the fifth field is the decimal representation of the local or remote port number. Together, IP address and port number completely specify a UDP or TCP connection (note that UDP "connections" consist of restrictions imposed upon which hosts may communicate with a socket). Either the IP address or the port part of the 5-tuple may be wildcarded, and if this is the case, is represented with an asterisk. The "Recv-Q" and "Send-Q" denote the number of bytes awaiting transmission on the connection, or awaiting processing by the NetROM process using TCP or UDP.

The *netstat* command also allows the NetROM user to monitor the activity level and type of four protocols in the TCP/IP protocol suite. These protocols are TCP, UDP, IP, and ICMP. A complete description of protocol statistics is beyond the scope of this document. However, they are generally either self-explanatory or only useful to experienced TCP/IP network administrators.

Finally *netstat* enables the user to display NetROM's routing table. This contains information used by NetROM to access IP hosts which are not on the local Ethernet. The routing table contains information about "routers," which are special network hosts that forward packets to computers with non-local addresses.

ping

Sends ICMP ECHO_REQUEST packets to network hosts.

Synopsis

ping **host_address [size [count]]**

Description

The *ping* command uses ICMP (Internet Control Message Protocol) to determine whether remote hosts are up and accessible. ICMP is a mandatory part of IP, and a host receiving an ECHO_REQUEST packet should respond with an ECHO_RESPONSE packet

The *ping* command actually creates a process which issues echo request packets. Upon receipt of a response packet, The *ping* process will print out the ICMP sequence number of the response, the host it was received from, and the size of the packet. The process will continue to send packets until it is killed with a SIGINT signal, unless a count value was specified on the command line. This signal can be issued from the controlling terminal using the "intr" character, usually ^C. Upon being killed, the *ping* process will print the number of echo requests it has sent, the number of responses it has received, and the ratio of the two expressed as a percentage lost.

The default ICMP datagram size is 64 bytes, but this value can be overridden on the command line. Note that the ICMP datagram size is not the same thing as the IP datagram size, or as the Ethernet packet size. If a count value is specified on the command, the *ping* process will send that many packets and then quit.

The *ping* process does not use keyboard input, so other commands may be entered while *ping* is running.

route

Manipulates information in NetROM's routing table.

Synopsis

route add **destination gateway** [**metric**]
route add default **gateway** [**metric**]
route delete **destination gateway**

Description

The routing table is used by NetROM for determining the path to nodes on networks to which NetROM is not directly attached. This might include hosts in another building, or in another country. The **destination** parameter is the IP address of the remote host with which NetROM will be communicating. The **gateway** parameter is the IP address of an intermediate host which will be responsible for forwarding packets sent from NetROM onward in their path to the remote host. The **metric** parameter is an indication of how "hard" it is to reach the destination via the gateway. Generally routing metrics are interpreted as a "hop count," which is the number of gateways between NetROM and the destination. All IP addresses should be given in standard dotted-decimal notation.

It is possible to assign NetROM a default route; this is the address of a computer on the local subnetwork to which NetROM will send packets destined for destinations on unknown networks. The default route can be set by invoking the *route* command with the *default* keyword, or by specifying a destination with IP address "0.0.0.0."

Note

On TCP/IP networks, there may be more than one route to a given destination, so *both* the destination and the gateway are required to fully specify a route. NetROM's current routing table can be displayed using the *netstat* command.

---

slip

Attaches or detaches a serial line to the Serial Line IP (SLIP) interface. The SLIP interface may be attached to either serial port of NetROM.

Synopsis

slip attach **port**
slip detach **port**

Description

The *slip attach* command designates which serial port the SLIP connection uses as its communication path. The **port** parameter may be either a 1 or a 0. A 1 indicates the SLIP connection should run over the remote port, while a 0 indicates the SLIP connection should run over the console port. When trying to establish a SLIP link with another computer the first step (after all cabling has been performed) is to issue the *slip attach* command.

The *slip detach* command removes a port from use by the SLIP interface. It should be entered when the SLIP connection is no longer needed.

Target Interface Commands

NetROM's target interface commands allow the NetROM user to download images to emulation memory, verify the images if desired, establish "consoles" with the target, and reset the target. These commands include *newimage*, *serialcons*, *tgtcons*, and *tgtreset*.

fill

Allows NetROM users to initialize emulation memory to arbitrary values.

Synopsis

fill **value** [**podgroup** | *dpmem*]

Description

This command fills the emulation space of one of NetROM's pod groups with a known value. This value is specified as an 8-bit hexadecimal number given by the **value** parameter. The optional **podgroup** parameter indicates which pod group's emulation memory should be filled. If omitted, the default podgroup is assumed. If the optional *dpmem* keyword is used instead of a pod group number, NetROM's dualport RAM (used for passing messages to the target system) will be filled with the value pattern instead of the whole pod default group.

newimage

Downloads a file into emulation memory.

Synopsis

newimage [**filename**] [*type*={*binary*|*srecord*|*intelhex*}] [*base*=**baseaddr**] [*offset*=**offset**] [*group*=**podgroup**] [*fillpattern*=**fillvalue**] [*host*=**ipaddr**]

Description

The *newimage* command allows the NetROM user to download, with TFTP, an image into ROM emulation memory. The command uses the NetROM environment variables to provide default values for all of the optional parameters listed above. However, the environment variables may be overridden, if desired. Note that there is no white space surrounding the equals signs when overriding defaults.

NetROM resolves address fields in Intel Hex and Motorola S-Record files using the base address of the destination pod group as a reference. The base address will be subtracted from the address given in the hex file when determining where in emulation memory to load a record. For example, if the base address of the target pod group is 0x40000 and a record's address field indicates address 0x40010, the record's data will be loaded at offset 0x10 into emulation memory. The base address for the default pod group is given by the "groupaddr" environment variable. If desired, this value can be overridden using the **baseaddr** parameter. Note that since binary files do not have an address field, they are always loaded at the beginning of emulation memory (unless the **offset** parameter is used, as described below).

It is possible to have an offset added to the destination address of a record, after it has been parsed and adjusted for the pod group's base. For example, if a target pod group's base address is 0 and records in a file are addressed beginning at 0, but the file is *really* located for address 0x100, setting the **offset** parameter to 0x100 will cause 0x100 to be added to the addresses of all records. This parameter can be used to control the load address of binary files, since they do not have address fields.

Attempts to program addresses outside of pod group emulation space will simply be ignored, but a warning message will be displayed after the download is complete.

The **fillvalue** parameter overrides the environment "fillpattern" variable. It may be set to any 8-bit value, or to *none*. If a fill pattern is specified, the entire target podgroup's emulation memory will be set to that value prior to downloading. If multiple image files are downloaded into emulation memory, it is important to set the fill pattern to *none* after the first download.

When issued with no arguments, the *newimage* command performs the following actions: it concatenates the "loadpath" and "loadfile" environment variables to get a root-specific path to the file to be downloaded. It uses the "filetype" environment variable to determine whether to expect a binary, Intel Hex, or Motorola S-Record file. The "host" variable determines the address of the TFTP server for the Ethernet, and the "podgroup" variable determines which of the four possible podgroups will be downloaded. NetROM then contacts the server, requests the file, and downloads it into emulation memory, parsing the file format as necessary. Note that the pod group and server address may be overridden using the **podgroup** and **ipaddr** parameters, respectively.

The *newimage* command disables target access to all emulation pods for the duration of the download. If emulation was on, NetROM will turn it off for the download and back on when the download is complete. It may be necessary to reset the target with the *tgtreset* command after a download.

When specifying the file to be downloaded, the file is assumed to be in the "loadpath" directory unless its name begins with a '/'. The '/' character denotes a root-specific filename and overrides the "loadpath" variable.

Pod groups are specified by number, not by name. Server addresses are denoted using standard dotted-decimal notation. Filetype is given as "binary," "intelhex," or "srecord," exactly as in the environment variable.

The *newimage* command may be issued as part of a batch file (see the *batch* command) if desired.

Example

Assume we are downloading two files into the default pod group's emulation memory. The "fillpattern" environment variable is *none*, the "filetype" variable is *srecord*, and the "groupaddr" variable is 0x40000. The first file is located at 0x40000 and the second file is a binary file which goes at 0x40100.

```
NetROM> newimage file1.hex fillpattern=ff
NetROM> newimage file2.bin offset=100 type=binary
```

These two commands load the S-Record file first, then the binary file. Note that the binary file overlays any data that was in the S-Record file starting at address 0x100. Prior to loading the first file, all of the default pod group's memory was loaded with 0xFF.

See Also

"verify" environment variable, page 4-123
"groupaddr" environment variable, page 4-111
"fillpattern" environment variable, page 4-110
tgtreset command, page 4-30
di pgconfig command, page 4-67
set emulate off command, page 4-39
set emulate on command, page 4-39

Note

A user may experience problems resulting from the target system's control of FLASH ROM write-enable lines. Some target systems may allow the Write Enable signal to their ROM sockets to "float." If a true "ROM" were plugged into the socket this would not be an issue, since the ROM ignores that signal. However, NetROM allows writes to emulation memory, so a floating write line can cause random changes to emulation memory. To disable the emulation pod's write signal, set the *groupwrite* environment variable to "readonly," or specify a read-only pod group if using the *set pgconfig* command.

serialcons

Allows NetROM users to create a "console" to a non-target system in environments wherein neither the debug path nor the console path use NetROM's target serial port.

Synopsis

```
serialcons
```

Description

The *serialcons* command allows NetROM users to make use of NetROM's target serial port, even when the target itself does not use it. To see how this may be useful, consider that some target systems may be plug-in boards which are used in some larger system. An example of this might be a board that does I/O for a standalone computer such as a terminal concentrator. The target board in this case might not have a serial port of its own, so during development the engineer using NetROM might use the "readwrite" or "readaddr" console and debug paths. This would cause the *tgtcons* command, and any remote debuggers being run, to use one of the dualport RAM mailbox protocols to communicate with the target. However, the target's "host" computer might have a serial port, and in this case the *serialcons* command would allow the engineer to use NetROM to communicate with both the host system *and* the target!

The *serialcons* command will not work unless neither the "debugpath" nor the "consolepath" environment variables is set to "serial." Remember that changes in these environment variables do not take effect until the target system is reset. If there is a conflict with the environment variables, an error message will be printed. In environments in which a serial port will be used to communicate with the target, the *tgtcons* command should be used instead.

To exit from a *serialcons* session, use the controlling terminal's "eof" character. The default eof character is ^D. The *stty* command can be used to display and set the eof character.

See Also

tgtcons command, page 4-28
"debugpath" environment variable, page 4-106
"consolepath" environment variable, page 4-104
stty command, page 4-96

tgtcons

Establishes a console session with the target system.

Synopsis

tgtcons

Description

The *tgtcons* command allows NetROM users to establish a console session with the target system, regardless of the mechanism used to implement the console path. The console path, which runs between the NetROM user on the host system and the target system to which NetROM is attached, has two segments. The first segment connects the host system and NetROM. This is generally a TELNET terminal session on NetROM but can be a "dumb" terminal connected to NetROM's console serial port, or a direct TCP connection to NetROM's user interface port. The second segment is between NetROM and the target. This can be either an RS-232 serial connection using NetROM's target serial port, or one of two emulation RAM mailbox protocols. These protocols are based on the target's ability to write emulation memory, as well as read it. The console path between NetROM and target system is selected by the "consolepath" environment variable.

When the *tgtcons* command is issued, NetROM begins to forward keystrokes received from the host side of the connection to the target, and data received from the target side to the host. The effect is the NetROM terminal session under which the command was issued becomes a session directly between the host system and target.

To exit from a *tgtcons* session, use the controlling terminal's "eof" character. The default "eof" character is D. If the console path uses the serial port, and since the default "eof" character may be "special" for the target, it is possible to re-map the "eof" character to another control character. The *stty* command can be used to display and set the "eof" character.

It is possible to send RS-232 BREAKs to the target. NetROM monitors *tgtcons* sessions for "intr" characters, and if the console path uses the serial port, it sends a BREAK to the

target. If NetROM's "intr" character is used by the target, simply change it to something else using the *stty* command, as explained for the "of" character, above.

Operation Change

In NetROM versions 1.2.7 and earlier, this command sent a "BREAK" through the target serial port on the invocation of the command. For versions after 1.2.7, the "BREAK" has been eliminated.

See Also

tgtcons command, page 4-28
"debugpath" environment variable, page 4-106
"consolepath" environment variable, page 4-104
stty command, page 4-96

tgtreset

Resets the target processor.

Synopsis

```
tgtreset
```

Description

When the *tgtreset* command is issued, NetROM performs the following actions:

1. It applies the reset pulse to the default *reset* command signal pinout (command signal 0).
2. It locks the target out of emulation memory. This has the same effect as the *set emulate off* command, and will cause the target processor to read garbage from emulation memory. This step is necessary to allow NetROM to reset the contents of emulation memory mailboxes without interference from the target.
3. NetROM resets emulation memory mailboxes and pointers, if they are being used for communication with the target.
4. NetROM unlocks target emulation memory. This has the same effect as the *set emulate on* command.
5. NetROM deasserts the reset pulse to the target.

The *tgtreset* command must be invoked after changing the environment "consolepath" or "debugpath" variables. This synchronizes the change of communication protocols between the target and NetROM with both parties involved. Note that *tgtcons* sessions do not need to be restarted after resets, even if the target-to-NetROM communications paths have been changed.

See Also

"tgtcons" command, page 4-28
"consolepath" environment variable, page 4-104
"debugpath" environment variable, page 4-106
set emulate off command, page 4-39
set emulate on command, page 4-39

Process Control Commands

The process control commands allow NetROM users to determine the state of processes running on NetROM, and to influence their execution using signals. These commands include *kill* and *ps*.

kill

Sends a signal to any process running on NetROM.

Synopsis

kill **signal pid**

Description

The *kill* command allows a process to send a signal to any other process, even one running under another controlling terminal.

The **signal** parameter is the signal number to be sent to the process. Processes may block or ignore any signal other than the SIGKILL signal. Valid signal values and their meanings are shown in Table 4-1.

The **process** parameter is the process id to which the signal is sent. Process ids may be obtained via the *ps* command.

ps

Displays the current status of processes running on NetROM.

Synopsis

```
ps [ -s | -p | -i ]
```

Description

Process status information is useful for determining whether processes are running normally. The *ps* command allows the user to determine whether or not processes are waking up periodically, what their current state is, what is the process group to which they belong (a process group is a set of processes sharing a single controlling terminal), and other information not generally useful to most users.

Without any arguments, *ps* displays the process' pid, name, current status, current number of wakeups, stack usage, and, if the process is sleeping, what its wakeup condition is. The other arguments are interpreted as follows:

-s prints a bit more information about the stack space allocated to each process, and about its process group.

-p prints a bit more information about the process itself. This is not useful to most users.

-i prints information about signals pending, blocked, and ignored on each process. It also displays the process group information.

Most users will use *ps* simply to determine the process id of processes they wish to kill; see "kill" on page 4-32. Processes will generally be *sleeping* unless they have work to do, so a process' wakeup count is a good way to determine its activity level. Some processes, such as the kernel, are constantly active so they are always in a *yielding* or a *ready* state. The "Kernel" process is special; its wakeup count is always zero despite its being constantly active.

Set Commands

NetROM maintains two kinds of state variables, each accessed by its own set of commands. NetROM *environment variables* have two important characteristics: they are independent of processes and they are frequently accessed. Being independent of processes means that they affect all processes, regardless of which process changes them. Being frequently accessed means that NetROM users want to display or change them relatively frequently.

The other kind of state variable, *generic variables*, are a catch-all for variables which are not environment variables. These variables are either process-specific or rarely used. The distinction between environment variables and generic variables is rather hazy, but will begin to make sense after you begin using NetROM.

The *set* command sets or modifies various generic NetROM state variables. State variables, which can be set with the *set* command, can be displayed with the *di* command. Table 4-5 summarizes the *set* command.

Table 4-5 Arguments to the *set* Command

Argument	Description
?	Displays arguments to the set command.
consecho	Enables or disables echoing console data passed to the target system.
debugecho	Enables or disables echoing debug data passed to the target system.
dplocation	Sets the location of the dualport pod for a large ROM.
emulate	Enables or disables target system access to emulation memory.
help	Displays arguments to the *set* command.
loadecho	Enables or disables debug information on downloads.

Table 4-5 Arguments to the *set* Command (Continued)

Argument	Description
pgconfig	Configures a pod group.
pgname	Assigns a name to a pod group.
podmem	Sets values in emulation memory.
prompt	Sets the session prompt.
raconfig	The number of target accesses expected when reading a read-address byte.
rawrites	Enable or disable target system requests that NetROM set a byte.
romupgrade	Installs a new version of the NetROM program.
tgtctl	Turns on or off command signals to the target system.
udpsrcmode	Set the state of the UDP source address variable.
username	Enable an advisory lock on the NetROM unit.

set consecho

Enables or disables echoing of console path data on the NetROM console port.

Synopsis

set consecho { *on* | *off* }

Description

Console echoing is meant to be used as a debug tool in cases where the host system is having trouble with its console path connection to the target. When console echoing is on, console data which is received from the target is echoed to NetROM's console port before being forwarded to the host system, and console data received from the host system is echoed to NetROM's console port before being forwarded to the target. If multiple target console sessions are active, data received from any of their host connections is echoed, but data received from the target is only echoed once.

In order to use console echoing, it is necessary to have a "dumb" terminal connected to NetROM's console serial port. This terminal will be able to issue commands to NetROM, just as any TELNET session can. If your terminal does not seem to be communicating with NetROM, you may need a null modem.

If no terminal is connected to the console port and console echoing is turned on, nothing will appear to happen. However, data will still be echoed out the port; this may cause a slight reduction in response time on the console path as perceived by the host system and the target.

Console echo can be enabled or disabled from any NetROM terminal session.

See Also

di consecho command, page 4-56

set debugecho

Enables or disables echoing of debug path data on the NetROM console port.

Synopsis

set debugecho { on | off }

Description

Debug echoing is a debug tool for cases in which the host system has trouble with its debug path connection to the target. When debug echoing is on, debug data received from the target is echoed to the console session that issued the `set debugecho` command before it is forwarded to the host system, and debug data received from the host system is echoed to NetROM's console port before it is forwarded to the target. If the environment variable "debugpath=serial" is set, data is displayed only on the NetROM console serial port.

Debug echo can be enabled or disabled from any NetROM terminal session.

See Also

di debugecho command, page 4-37

set dplocation

Indicates which NetROM pod will contain the dualport RAM on "large" ROMs.

Synopsis

```
set dplocation { high | low }
```

Description

This command controls where NetROM puts dualport RAM on "large" ROMs. A "large" ROM is one which has more than 256 Kwords; the 27c040 is an example of such a device. Large ROMs are implemented using an active pod cable which combines two of NetROM's emulation pods into one ROM plug. If the NetROM pods are used sequentially to create a 'longer' ROM, then NetROM pod 0 (and dualport RAM) can be in high memory or low memory on the ROM.

This variable should be set to agree with a jumper on the active pod. This variable controls where data is loaded into emulation memory on long ROMs, so it may be necessary to redownload after changing it.

See Also

di dplocation command, page 4-58

set emulate

Enables or disables target access to emulation memory on all pods.

Synopsis

```
set emulate {on | off }
```

Description

Due to the asynchronous nature of target system access to emulation memory, it is sometimes necessary to disable target access entirely. Target access is asynchronous, because ROM devices do not use a clock input. The target asserts an address on the ROM address lines, waits a certain number of clock cycles for data to stabilize on the data lines, and then samples the data. If the target tries to access emulation memory while NetROM is accessing it, the target will read garbage. Likewise, if the target is in the process of accessing emulation memory, NetROM accesses may be held off indefinitely.

While NetROM does not generally access emulation memory, it may occasionally want to do so. For example, it may need to download a new emulation image or display the contents of emulation memory. Some NetROM commands, such as *di podmem,* require that the user explicitly disable emulation. Others, such as *newimage*, will automatically disable emulation (and re-enable it when done).

The *set emulate* command allows the user to explicitly enable or disable target access to emulation memory.

Note

If the target system is connected to a writable pod group, powering the target on or off with emulation on may corrupt the emulation memory. This is due to possible noise on the target's write line(s).

See Also

di emulate command, page 4-61

set pgconfig

Completely configures a pod group for emulation.

Synopsis

set pgconfig **groupnum romtype tgtaddr podorder**
{ *readonly* | *readwrite*}

Description

The *set pgconfig* command is used to completely define a podgroup prior to downloading it with an emulation image. This command is probably most useful in environments in which NetROM is emulating more than one group of target ROMs. Because it is simpler and generally more convenient to configure pod groups using environment variables than with the *set pgconfig* command.

An example of a multiple ROM-group target would be one in which one set of ROMs holds an executable image and another a graphics table. The engineer using NetROM would then choose the most-often-updated group of ROMs and configure it with environment variables, and configure the other group with the *set pgconfig* command.

The **groupnum** parameter indicates which pod group is being configured. If the "special" pod group named by the "podgroup" environment variable is being configured, NetROM will change environment variables to agree with the command-line specification for this command. Refer to "Pods and Pod Groups" on page 2-2.

The **romtype** parameter is the name of the ROM type being emulated. Valid ROM types are given by Table 4-11.

The **tgtaddr** parameter is the 32-bit base address of the pod group as seen by the target. This value is used by the *di podmem* command to display emulation memory with the same addresses as are used in a map file produced by a compiler.

The **podorder** parameter specifies which emulation pods are to be used in the group, and in what order they emulate target bytes. The format for podorder parameters is described more

fully in "podorder" on page 4-117. If the specification of pods in the **podorder** parameter conflicts with pods in use by the pod group named by the "podgroup" environment variable, the command will fail with an error message.

Some target systems are unable to write to their ROM space only because the system designer omitted providing a write signal to the ROM memory. This is a logical thing to do, because it is not possible to write a ROM. Some of these target systems may be able to write ROM emulation memory on NetROM if a write signal were provided. Such target systems may use NetROM's external write line to connect to the processor. The *readonly* and *readwrite* keywords indicate whether or not the target system should be allowed to write emulation memory for a given pod group, using the external write line.

Examples

set pgconfig `0 27c020 bfc00000 0:1:3:2` *readonly*
Configures pod group 0 to emulate 27c020 ROMs. The pod group starts at target address 0xBFC00000, and emulates 32-bit words. Note that pod 0 emulates byte 0 of the word, and that pod 1 emulates byte 1, but that pod 3 emulates byte 2 and pod 2 emulates byte 3. The target system will not be allowed to write emulation memory

set pgconfig `0 27c010 0 1:0-3:2` *readwrite*
Configures pod group 0 to emulate 27c010 ROMs. The pod group starts at target address 0x00000000, and emulates 16-bit words. Pods 0 and 1 emulate one set of words, and pods 2 and 3 emulate another, which begins where the words emulated by pods 0 and 1 leave off. Note that pods 0 and 2 emulate byte 0 of the word, and that pods 1 and 3 emulate byte 1. The target system will be allowed to write emulation memory

set pgconfig `0 27c010 0 1-0-2-3` *readonly*
Configures pod group 0 to emulate 27c010 ROMs. The pod group starts at target address 0x00000000 and emulates 8-bit words. Since 27c010 ROMs have 128 Kilobytes, each pod emulates 128 Kwords, where each word is eight bits wide. Note

that the pod group as a whole emulates 512K of consecutive words, where pod 1 emulates the first 128K, pod 0 the second, pod 2 the third, and pod 3 the fourth. The target system will not be allowed to write emulation memory using the external write line.

See Also

"podgroup" environment variable, page 4-116
"romtype" environment variable, page 4-120
"groupaddr" environment variable, page 4-111
"romcount" environment variable, page 4-119
"wordsize" environment variable, page 4-126
"podorder" environment variable, page 4-117
di pgconfig command, page 4-67
di podmem command, page 4-68
set pgname command, page 4-43
setenv command, page 4-101
printenv command, page 4-102

set pgname

Assigns a name to a pod group.

Synopsis

set pgname **namestring** [**podgroup**]

Description

The *set pgname* command assigns a name to a pod group. This pod group must have either been configured using environment variables or with the *set pgconfig* command. Pod group names are optional and exist for the convenience of the NetROM user. They are essentially mnemonics to help the NetROM user remember what the pod group is emulating, if more than one pod group is in use.

The **namestring** parameter is the name being assigned to the podgroup. The podgroup to which the name is assigned is defaulted to that named by the "podgroup" environment variable. This default can be overridden by the **podgroup** parameter.

The *set pgname* command will not work on pod groups which have not yet been configured.

See Also

set pgconfig command, page 4-40
di pgconfig command, page 4-67

set podmem

Allows NetROM users to selectively set values in emulation memory.

Synopsis

set podmem **address value**

Description

The *set podmem* command allows NetROM users to set values in emulation memory. The **address** parameter determines where to set the value, and **value** is the 8-bit quantity being written to memory. NetROM uses the **address** parameter to determine which pod group will be affected by the write operation.

set prompt

Changes the prompt for the NetROM terminal session which issued it.

Synopsis

set prompt [*-d*] **promptstring**

Description

The *set prompt* command changes the prompt for the current NetROM terminal session to the value given by **promptstring**. The new prompt cannot contain any white space; that is, it must a single "word." If the optional *-d* flag is used, NetROM will set the default prompt for all subsequent terminal sessions as well as the prompt for the current terminal session.

set raconfig

Used to configure the number of accesses NetROM will expect that the target make in the course of reading a single byte of data from the interrupt area in dualport RAM.

Synopsis

set raconfig **numaccesses**

Description

This command is used to describe the target system's pattern of accesses to emulation memory. This information is important when the dualport memory mailbox protocols are used to pass data between NetROM and the target system. Some targets may execute multiple read cycles to read a single byte; for example, a processor with a 32-bit data bus which is reading opcodes from a single ROM may have memory interface hardware which performs 4-byte read cycles in order to present the processor with a 32 bit opcode.

On some of these target platforms, the memory interface hardware performs four read cycles, *even when the processor only requests a single byte*! This confuses the dualport protocols, which require that a particular byte be read by the target to send an 8-bit value. The **numaccesses** parameter to this command describes the number of accesses to pod 0 the target processor will make in the course of making a *one-byte access*.

For example, if a 32-bit processor attempts to read a single byte of ROM but its ROM interface hardware performs four read cycles (to make a 32-bit word) in order to present the processor with the single byte it requested, then **numaccesses** for this target is four. If on a different target, the 32-bit processor requests a single byte from ROM and its interface hardware performs a single read cycle to fetch the byte, then **numaccesses** is one for the target. Finally, if a 32-bit processor requests a byte for a 16-bit ROM word composed of two 8-bit ROMs, one of which is NetROM's pod 0, and the interface hardware performs two read cycles on pod 0 and two ready cycles on the other ROM (to assemble a 32-bit word),

then **numaccesses** for this target is two. This is because only two accesses were made to pod 0 in the course of requesting a single byte, even though four accesses to ROMs were made overall.

Note

Read-address configuration is only important when using the read-address dualport protocol; when using the readwrite dualport protocol there is only one interrupt-causing address defined.

set rawrites

Used to enable or disable target requests that NetROM write emulation memory.

Synopsis

```
set rawrites { on | off }
```

Description

This command allows NetROM users to have the target system request that NetROM set a byte of memory in emulation space. Enabling read-address writes is equivalent to partially enabling the readaddr path to the target. However, if rawrites are on, neither the console path nor the debug path is required to use the readaddr protocol. (Note that neither path may use the readwrite protocol; however, if the target can write its own emulation memory, there is no reason for it to use read-address writes.)

If rawrites are enabled, and one of the target paths is set to readaddr, the entire readaddr protocol will be used. There is no need to disable rawrites prior to enabling the readaddr protocol. If the readaddr protocol is disabled and rawrites are still on, the set memory portion of the read address protocol will remain enabled.

As when setting the console or the debug path, changes to the rawrites variable require the target to be reset with the *tgtreset* command before they take effect.

set romupgrade

Initiates the download of a new NetROM operating system image.

Synopsis

set romupgrade [*ramimage*=**ramname**] [*romimage*=**romname**] [*host*=**ipaddr**]

Description

The *set romupgrade* command is used to update NetROM's ROM-based operating system image. It should only be used when a new system is distributed. When upgrading, Applied Microsystems will make available two binary files, *nsXXXXXX.bin* and *netrom.bin*; *nsXXXXXX.bin* is the new system image and *netrom.bin* is a RAM-based download program. The name of the new system image is based upon the Ethernet address of the NetROM unit being upgraded. The last six characters of the file name are the last six characters of the unit's Ethernet address. To perform the upgrade, these two files should be placed in the TFTP directory named by your "loadpath" environment variable, on the host named by your "host" environment variable. Then, invoking *set romupgrade* command with no arguments will cause *netrom.bin* to be loaded into RAM and control transferred to it. *Netrom.bin* will download *nsXXXXXX.bin* into FLASH ROM memory in your NetROM unit automatically. When the download is complete, you should reset your NetROM unit.

The optional settings allow users to control the paths to the RAM-based image which will reprogram the ROMs, the new ROM system image, and the IP address of the TFTP server to contact for both images. The default ROM image name is determined by the NetROM unit's Ethernet address; for example, if the unit's hardware address is "00:02:f4:00:00:24" its new system image should be named *ns000024.bin*. The default RAM image name is *netrom.bin*; and the default server address is given by the "host" environment variable. If the image names are not root-specific, they are assumed to be in the directory given by the "loadpath" environment variable.

Note

If you initiate the download from a Telnet session, the unit will appear to reset when the download of *netrom.bin* is complete. However, the unit has simply transferred control to a RAM-based image, to which you can also Telnet. The complete FLASH reprogramming may take as long as 5 minutes. We recommend you monitor the progress of the download on the NetROM console.

If you do not have a serial console handy, the download has completed after:

1. *netrom.bin* has been downloaded and jumped to. At this point your telnet session will stop responding. You should exit it and re-telnet to NetROM.
2. ATFTP client has been created, run for a while, and exited. You can see this process using the *ps* command. NetROM is verifying that *nsXXXXXX.bin* can be downloaded.
3. There is a period in which telnet response seems sluggish, NetROM's heartbeat LED is *very* slow, and there is no TFTP client present. At this point, NetROM is erasing its FLASHes.

Note

Do not reset or power cycle NetROM after this point.

4. A new TFTP client has been created, run for a while, and exited. At this point, NetROM is downloading its new image, *nsXXXXXX.bin* and is programming it into the FLASHes.

Note

Do not reset or power cycle NetROM during this process.

5. NetROM's Ethernet transmit LED is not longer flashing, there is no TFTP client running, and the heartbeat LED is flashing quickly (at its normal speed). At this point it is safe to reset your NetROM with the *reset* command.

   NetROM reboots the system automatically after a successful upgrade is completed.

Note

Do not attempt to copy system ROMs. ROM-based images intended for one unit will not work on a different unit, unless you have a multi-unit upgrade license.

set tgtctl

Used to control NetROM's control signal outputs.

Synopsis

```
set tgtctl signum {on | off} [millisec_interval]
set tgtctl signum {on | off} [toggle]
set tgtctl signum {on | off} [rx] [ack]
```

Description

This command asserts or de-asserts one of the control signals on NetROM's front panel. When *on*, these signals are connected to ground (*low true*); when *off*, these signals do not assert or draw current to or from the target. If the optional **millisec_interval** parameter is present, the signal will be asserted *on* or *off* as specified with that period. When not asserted, the signal will have its alternate value. The granularity of the interval is 5 milliseconds, values will be rounded down to these increments. The timer will run until the signal is turned on or off without a new interval.

When the **toggle** parameter is present, the signal will be asserted briefly, then return to its alternate state. This can be used to cause an interrupt to the target system. This use of the target control signal requires hardware support from the target system.

The third formulation of this command asserts the signal to the specified value when data is passed to the target using one of the dualport emulation memory protocols. This signal can be attached to the target system, causing an interrupt to the target when data is ready to be read. If the **ack** parameter is specified, NetROM won't trigger a subsequent interrupt if the previous one has not been acked.

This use of the target control signal requires hardware support from the target system.

set udpsrcmode

Used to enable connectionless debug sessions.

Synopsis

set udpsrcmode { on | off }

Description

This command controls NetROM's treatment of UDP-based debug sockets. When enabled, NetROM prepends the IP address and UDP port number of the packet being sent to targets which use dualport RAM for their debug paths. Similarly, data received along the dualport paths is assumed to have a 32-bit IP address followed by a 16-bit port number prepended to actual data. These values will be sent and interpreted in network byte order. This mode allows target systems to specify the destination address of packets generated by the target's debugger interface.

Since a start-of-packet/end-of-packet sequence is not defined for the serial interface, UDP source mode cannot be used for the serial debug path. UDP source mode is only used on the debug path; UDP header information is only prepended to data received on the NetROM debug port. Source address information is not added on TCP-based debug sessions, nor on console sessions.

If UDP source mode is turned on while a debug connection is active, the target must be reset with the *tgtreset* command before UDP source mode is actually enabled.

Display Commands

The *di* command displays various generic NetROM state variables, various NetROM statistics, and target state information. State variables which are set with the *set* command can be displayed with the *di* command. Statistics can be displayed for NetROM's target and console serial port UARTS, NetROM's Ethernet interface, and for memory usage. Table 4-6 summarizes the *di* command.

Table 4-6 *di* Command Arguments

Argument	State or statistics displayed
?	List of *di* arguments and what they display.
consecho	Console echo state, on or off.
debugecho	Debug echo state, on or off.
loadecho	Load echo state.
dplocation	Location of the dualport pod for a large ROM.
dpmem	Contents of dualport RAM.
dpstats	Statistics for dualport protocols.
emulate	Target access to emulation memory.
help	List of *di* arguments and what they display.
lanceha	NetROM's Ethernet address.
ledmap	Mapping between NetROM status signals and back panel LEDs.
lstats	Ethernet statistics.
memstats	Memory use statistics.
modules	Names of optional RAM modules loaded.
pgconfig	Pod group configurations.

Table 4-6 *di* Command Arguments (Continued)

Argument	State or statistics displayed
podmem	Contents of emulation memory.
raconfig	Number of target accesses expected when reading read-address byte.
rawrites	Current state of the rawrites variable.
tgtctl	State of NetROM's command signals.
tgtstatus	State of NetROM's status signals.
uart	Statistics for NetROM's serial ports.
udpsrcmode	Current state of the UDP debug source address variable. See *set uadpsrcmode*.
uptime	Time since the last system reset.
username	User name used for advisory login locks.
version	Software version number for NetROMs operating system.

di consecho

Displays whether console echoing is turned on or off.

Synopsis

```
di consecho
```

Description

di consecho prints to the screen the current state of NetROM's console echo variable. To change the variable, use the *set consecho* command.

See Also

set consecho command, page 4-36

di debugecho

Displays whether debug echoing is turned on or off.

Synopsis

```
di debugecho
```

Description

di debugecho prints to the screen the current state of NetROM's debug echo variable. To change the variable, use the *set debugecho* command.

See Also

set debugecho command, page 4-37

di dplocation

Displays whether the dualport RAM location is high or low.

Synopsis

```
di dplocation
```

Description

di dplocation displays the setting of the dplocation variable, which controls where NetROM puts dualport RAM on "large" ROMs. A "large" ROM is one which has more than 256 Kwords; the 27c040 is an example of such a device.

See Also

set dplocation command, page 4-38

di dpmem

Displays the contents of the dualport RAM used to pass messages between NetROM and the target.

Synopsis

di dpmem **dpoffset nbytes**

Description

The *di dpmem* command helps the NetROM user to debug the target's dualport mailbox code, which is used to pass messages between NetROM and the target. Pod 0's dualport RAM, which can be accessed simultaneously by both the target and NetROM, is described in detail in Chapter 7. This command is provided as a convenience to allow programmers to examine the mailbox structures in dualport RAM without having to know where in pod 0 the RAM is mapped.

This command displays **nbytes** bytes of dualport RAM, starting at offset **dpoffset** from the start of the dualport area. The same **dpoffset** value can be used regardless of where the dualport RAM is mapped within pod 0, and only dualport RAM data will be displayed, regardless of the word width of the pod group of which pod 0 is a part.

See Also

di podmem command, page 4-68
"Dualport Emulation Memory" on page 7-4
"The Dualport Message Structure" on page 7-6

di dpstats

Displays statistics for the dualport protocols, if any, used to pass data between NetROM and the target system.

Synopsis

di dpstats

Description

This command displays statistics about the dualport protocol used to forward data between NetROM and the target system. For the read-address protocol, statistics include the number of bytes sent to and received from the target, the number of messages sent to and received from the target, and the number of various "out-of-band" control characters received from the target. Although the read-address protocol will not drop characters received from the target, a count is made of the number of times the NetROM interrupt handler's input queue filled completely before NetROM was able to service the incoming characters.

Statistics for the readwrite protocol are simpler; they include the number of bytes and messages sent to and received from the target, and a count of error conditions, such as transmit timeouts, occurring on both sends and receives.

All statistics are reset by the *tgtreset* command, page 4-30.

di emulate

Used to determine whether ROM emulation is turned on.

Synopsis

```
di emulate
```

Description

The *di emulate* command prints the current state of target image emulation.

See Also

set emulate command, page 4-39

di lanceha

Prints out the 6-octet address used by NetROM's Ethernet interface.

Synopsis

```
di lanceha
```

Description

The *di lanceha* command will print out the 6-octet address used by NetROM's LANCE Ethernet interface. This address will be displayed in colon-separated hexadecimal format. The *di lanceha* command is primarily useful for setting up host configuration files which will be used in address resolution at NetROM boot time.

di ledmap

Shows the mapping between NetROM's status signals and LEDs on NetROM's back panel.

Synopsis

di ledmap

Description

The *di ledmap* command shows the mapping between NetROM's status signals and LEDs on NetROM's back panel. Mappings are sorted by signal number, then LED number.

See Also

ledmap command, page 4-92

di lstats

Prints a summary of packet and error counters for NetROM's Ethernet interface.

Synopsis

```
di lstats
```

Description

The *di lstats* command displays a summary of packet and error counters for NetROM's LANCE Ethernet interface. A complete summary of these statistics is beyond the scope of this document, but they are generally either self-explanatory or useful only for detecting gross errors. The error counters should all be zero, or very low, during normal NetROM operation. High error counts may indicate a problem with NetROM's LANCE chip or a malfunctioning host on the Ethernet network.

di memstats

Displays current memory allocation statistics for NetROM.

Synopsis

```
di memstats
```

Description

The *di memstats* command allows the NetROM user to examine the availability of allocation memory within NetROM's operating system. This command is primarily used to detect pathological states during NetROM operation and is not useful during normal operation. NetROM maintains several pools of allocation memory; during normal operation there should always be memory available in each of them. This can be verified by examining the "free mbufs," "clfree," and "free blocks" fields in the memory statistic display. None of these values should be zero.

di modules

Displays the names of the optional RAM modules that have been loaded into NetROM.

Synopsis

```
di modules
```

Description

The *di modules* command allows the NetROM user to display the names of the optional RAM modules that have been loaded.

di pgconfig

Used to display the current pod group configurations.

Synopsis

di pgconfig [**podgroup**]

Description

The *di pgconfig* command allows the NetROM user to examine the current state of emulation pod groups. The command displays, in tabular form, the name, word size, ROM type, target address, pod order, and read/write characteristics, of all podgroups defined in the system. If a pod group is specified with the **podgroup** parameter, only the configuration for that group will be shown.

See Also

set pgconfig command, page 4-40
podgroup command, page 4-116

di podmem

Allows the NetROM user to examine the contents of emulation memory.

Synopsis

di podmem **tgtaddr nbytes**

Description

The *di podmem* command displays the contents of emulation memory. The **tgtaddr** parameter is the address, as seen by the target, at which to start dumping memory. The **nbytes** parameter is the number of bytes to dump. NetROM uses the **tgtaddr** parameter to determine which podgroup should be displayed.

Due to hardware restrictions imposed by the nature of ROM devices emulation must be turned off for the *di podmem* command to work. If emulation is on, it will print an error message.

See Also

set pgconfig command, page 4-40
set emulate command, page 4-39

di raconfig

Indicates how many target accesses NetROM should expect in the course of reading a single byte from the read-address interrupt area.

Synopsis

```
di raconfig
```

Description

This command displays the number of ROM accesses which the target processor is expected to make to pod 0 in the course of reading a single byte. For most targets, this value should be one. However, some target processors may have ROM interface hardware which performs multiple accesses to assemble a complete word, even though the processor has only requested that the interface read a single byte.

See Also

set raconfig command, page 4-46

di rawrites

Used to display the current state of the rawrites variable.

Synopsis

```
di rawrites
```

Description

This command is used to determine whether or not the target system is capable of using read-address requests to set emulation memory on NetROM. Read-address write requests use the read-address protocol to ask NetROM to modify its own emulation memory; this facility is useful for systems which are unable to write to their own ROM space.

See Also

set rawrites command, page 4-48

di tgtctl

Displays the current status of NetROM's target control signals.

Synopsis

di tgtctl

Description

This command displays the current state of NetROM's target control signals on the front panel. When *on*, these signals are connected to ground (*low true*); when *off* these signals do not assert or draw current to or from the target. The flags field is set to 'n' if no special processing has been assigned to the signal. A numeric value indicates that the signal will be asserted with a period, in milliseconds, equal to that value. If the flags field contains "RX", the signal will be asserted to the target whenever NetROM sends data to the target using one of the dualport protocols.

See Also

set tgtctl command, page 4-52

di tgtstatus

Displays the current state of the status signals on the NetROM front panel.

Synopsis

```
di tgtstatus
```

Description

The *di tgtstatus* command displays the current state of the status signals on the NetROM front panel. A disconnected signal will read as "off."

di uart

Displays statistics for NetROM's serial port UARTs.

Synopsis

di uart [**uartnum**]

Description

The *di uart* command displays statistics for NetROM's serial port UARTs. The statistics include transmit, receive, and error counts, as well as counts for various sorts of interrupts. This command is useful for checking the quality of serial links between NetROM and the target, or between NetROM and a "dumb" terminal. When invoked without arguments, the *di uart* command prints statistics for both UARTs. When invoked with the optional **uartnum** parameter, it will only print statistics for one port. A **uartnum** value of 0 indicates the console port and a value of 1 indicates the target port.

di udpsrcmode

Used to determine the state of the UDP source address mode variable.

Synopsis

```
di udpsrcmode
```

Description

This command prints the current state of the UDP debug source address variable, which controls whether or not data forwarded between the target system and the host along the debug path will have IP addresses and UDP port numbers prepended. If enabled, UDP source address mode allows the target system to determine which of possibly many sources is sending it data, and to specify to which of possibly many destinations its data should be forwarded.

See Also

set udpsrcmode command, page 4-53

di uptime

Displays the amount of time since the last system reset.

Synopsis

di uptime

Description

The *di uptime* command allows NetROM users to determine how long their NetROM system has been running. Time is displayed in days, hours, minutes, and seconds.

di username

Displays who, if anyone, has installed an advisory login lock on the NetROM unit.

Synopsis

di username

Description

The *di username* command allows NetROM users to determine who, if anyone, has installed an advisory login lock on the NetROM unit.

di version

Displays the software version number of NetROM's operating system.

Synopsis

```
di version
```

Description

The *di version* command displays the software version number of NetROM's operating system.

ROM Set Commands

For target systems which require large ROM address spaces or word sizes greater than 32 its, a new group of commands has been defined. These commands manipulate a multi-NetROM data structure called a "ROM set." When using ROM sets, one NetROM unit is designated the "master" and one or more other units are designated as "slaves." The master unit's responsibility is to provide a command line interface to the NetROM user such that it appears that the emulation memory of *all* units in the set are local to the master unit.

For example, assume a target system has a 64-bit word size and uses 27c020 ROMs. Then two NetROM units can be used to define a ROM set. One will emulate the least significant 32 bits of the word, and the other will emulate the most significant bits of the word. Download and display of emulation memory would take place on the master unit using the *newimage* and *di podmem* commands, exactly as if all of the emulation memory resided on the master unit. The console and debug paths would also pass through the master unit.

Another example might involve a target system which required 4 megabytes of emulation memory, made up of 27c020 ROMs. A ROM set using four NetROM units could be defined, where each unit emulated its own successive megabyte of ROM. Again, download and display of emulation memory would take place through the master unit.

Emulation using ROM sets has four distinct stages: ROM set definition, in which the pod orders and IP addresses of slave units are defined on the master; connection, in which the master unit makes TCP connections with all slave units and puts them into slave mode; emulation, in which image downloads and other communications with the target system are carried out as normal; and disconnection, in which the master unit releases slave units and disconnects from them. There are specific commands to accomplish each of these steps, as well as commands to display the current ROM set status.

Note that certain commands become restricted when a NetROM unit is in slave mode. For example, the *tgtreset* command is not allowed, nor is the *set emulate* command, nor are *setenv* commands which affect pod order, word size, or ROM count. This is because all of these functions are taken over by the master unit. For example, if the master unit receives the command *set emulate off*, emulation will be disabled on all slave units as well.

Table 4-7 *romset* Command Arguments

Argument	State or statistics displayed
?	Displays arguments to *romset* command.
clear	Clears current romset definitions.
connect	Connects to slave units and enters romset mode.
define	Defines the romset pod order.
disconnectt	Disconnects from slave units and returns to normal mode.
help	Displays arguments to *romset* command.
show	Displays the current romset configuration.
slaveaddr	Sets the addresses of slave units.
reset	Resets all slave units.

romset clear

Erases ROM set definitions.

Synopsis

romset clear [*podorder* | *slaveaddr*]

Description

The *romset clear* command erases current ROM set definitions. It should be used when changing the number of slave units currently configured. If the unit count remains constant, use the *romset define* or the *romset slaveaddr* commands instead. Note that this command cannot be used while the NetROM unit is in slave mode or is connected to slave units.

See Also

romset define command, page 4-82
romset slaveaddr command, page 4-85

romset connect

Causes NetROM to create TCP connections with slave units.

Synopsis

```
romset connect
```

Description

The *romset connect* command causes NetROM to connect to slave units. When the command is issued, the unit it is issued on becomes a ROM set master and the units it connects to are put into slave mode. The ROM set must be defined and slave unit addresses must be given before this command is issued.

See Also

romset slaveaddr command, page 4-85
romset disconnect command, page 4-83

romset define

Configures the pod orders of all units in the ROM set.

Synopsis

romset define **order-string**

Description

The *romset define* command configures the pod order of the ROM set master unit, as well as the pod orders of all slave units. The pod order syntax is similar to that of the *setenv podorder* command, with the addition that the pod order for each unit is enclosed within parentheses. The **order-string** for each unit may be separated by hyphens ('-') indicating that words do not span units, or by colons (':') to indicate a large word size. The master unit's pod order is always the first in the list. Currently the largest word size supported is 64 bits. Note that the number of slave units indicated must agree with the number specified in the *romset slaveaddr* command.

Examples

romset define `(0:1-2:3)-(0:1-2:3)`
This command defines a ROM set in which two units support four consecutive sets of 16-bit words.

romset define `(0:1:2:3):(0:1:2:3)-(0:1:2:3):(0:1:2:3)`
This command configures a ROM set in which four units support two consecutive sets of 64-bit words.

romset disconnect

Terminates the current ROM set connection.

Synopsis

romset disconnect

Description

The *romset disconnect* command causes the master unit to restore connected slave units to normal mode and closes its network connections with them. Note that the environment characteristics defined by the ROM set will remain in effect on all units.

romset show

Displays the current ROM set configuration and status.

Synopsis

romset show

Description

The *romset show* command displays the current ROM set state for the unit it is invoked on. The state information includes slave unit addresses and pod orders, whether the unit is connected or not, and whether or not the unit is in slave or master mode. The "word index" displayed is only used when emulating ROM words larger than 32 bits. Since each unit can only emulate 32 bit words, the word index indicates which 32-bit increment of a word is emulated by the unit this command is invoked on. The word index is not set directly, but is implied by the order-string given in the *romset define* command. Note that the ROM set master is always at word index zero.

romset slaveaddr

Used to assign the network addresses of ROM set slave units.

Synopsis

```
romset slaveaddr addr1 [addr2 ...]
```

Description

The *romset slaveaddr* command is used to assign the (IP) network addresses of ROM set slave units. The IP address of the master unit should *not* be included in the list. Up to 8 units may currently be specified. Note that the number of units indicated must agree with the number of slave units implied by the *romset define* command. There is a direct correspondence between the order of units named in the *romset slaveaddr* command and the units implied by the *romset define* command.

See Also

romset define command, page 4-82

romset reset

The *romset reset* command causes all slave units to reset themselves.

Synopsis

romset reset

Description

The *romset reset* command resets all slave units. The unit issuing the command must be the ROM set master. This command essentially causes all connected slave units to execute a *reset* command, but does not cause the master unit to reset. The master unit reverts to normal mode after issuing this command.

Miscellaneous Commands

NetROM provides several miscellaneous commands for the convenience of the user. These include *alias, batch*, *help*, *history*, *ledmap*, *loadmodule*, *reset*, and *stty*.

alias

Used to create and delete command "nicknames."

Synopsis

alias **[alias-name [alias-string]]**
alias -d **alias-name**

Description

The *alias* command allows NetROM users to create nicknames for commonly used commands. When invoked without arguments, it lists all defined aliases, with the *-d* flag, it deletes a defined alias. When invoked with the *alias-name* parameter but no *alias-string* parameter, it displays the alias defined for that name. If both an *alias-name* and an *alias-string* are defined, the command assigns the alias string to substitute for the alias name in command invocations.

Examples

The alias assignment

```
alias nb newimage type=binary
```

causes the command

```
nb myfile.bin
```

to be executed as if it had been entered

```
newimage type=binary myfile.bin
```

The alias assignment is deleted with

```
alias -d nb
```

Note

Aliases can be nested; that is, an alias can include another alias in its expansion. Also, defining aliases uses memory, so defining aliases excessively should be avoided. A pre-defined command name cannot be used as an alias; for example,

```
alias set di
```

will not work. Aliases are invoked only after a match with defined command names fails.

---

batch

Downloads and executes batch files containing one or more NetROM commands.

Synopsis

batch **filename** [**server**]

Description

The *batch* command enables NetROM users to execute many NetROM commands with one command-line invocation. These commands are read from a file residing on the TFTP server which NetROM uses to load new images; this is the file server named in the "host" environment variable. The format of the file is a series of NetROM commands separated by new lines and terminated with an *end* statement. A *begin* statement at the beginning of the file is optional but recommended. See "Batch Processing" on page 4-7 for an example of a batch file.

The **filename** parameter names the file containing NetROM commands. If the name is not root-specific; that is, if it does not begin with a '/', the **filename** parameter will be appended to the "batchpath" environment variable to produce a root-specific path on the server. The optional **server** argument allows the command issuer to override the default environment setting for the TFTP server.

All commands executed as a result of the *batch* command will be entered into the history buffer for the terminal session under which the command was issued. The *batch* command may be "nested"; that is, it may be executed from within a batch file.

help

Accesses NetROM's on-line help facility.

Synopsis

help [**command**]
? [**command**]

Description

The *help* command accesses NetROM's on-line help facility. When invoked without arguments, the command prints a listing of available commands. When invoked with the **command** argument, it prints information specific to that command. When the command is a "nested" one, such as *set*, *di*, *setenv*, or *printenv*, it will print a list of the commands which come under that heading. It is possible to get help on nested commands by specifying which specific command in the **command** parameter. For example, *help set emulate* will get help on the *set emulate* command. The question mark "?" is a shorthand equivalent of the *help* command.

history

Displays the contents of the history buffer for the current NetROM session.

Synopsis

```
history
```

Description

The *history* command displays the contents of the history buffer for the NetROM terminal session under which the command was issued. Commands are numbered within the history buffer, allowing them to be invoked by number or special character (e.g., !!) for history substitution. See "History Substitution" on page 4-6 for details on history substitution.

ledmap

Allows users to map NetROM's status signals to LEDs on the back panel.

Synopsis

ledmap set **signum lednum** [*hightrue*]
ledmap clear **signum**

Description

The *ledmap* command establishes a path between status signals (signum) connected to traces on the target system and the LEDs (lednum) on NetROM's back panel (see Figure 2-1). The *set* version of this command establishes the mapping, and the *clear* version deletes it. Each status signal may be mapped to only one LED, but more than one signal may be mapped to each LED. Note that LED 0 is NetROM's "heartbeat" LED; by default, it indicates that NetROM is active and gives some indication of load on the system. If LED 0 is mapped using this command, its heartbeat function will be disabled for the duration of the mapping.

The *hightrue* keyword inverts the "normal" sense of status signals, so that rather than being "on" when tied to ground (or "low") they become "on" when asserting current.

Note

Status signals are polled, so they do not latch target-side events on the traces to which they are connected.

See Also

di ledmap command, page 4-63

loadmodule

Loads NetROM's optional RAM modules.

Synopsis

loadmodule **[filename** | *init]*

Description

This command downloads a RAM-based module that implements or extends NetROM features and commands. Normally, **filename** is appended to the string given by the "batchpath" environment variable; however, if the filename begins with a slash (/), the "batchpath" environment variable will not be used. (*init* initializes the module extension table; it is for Applied Microsystems development use only).

Note

The RAM module software is loaded into specific addresses in DRAM. We recommend that the file be stored in the same directory as your startup.bat file, but the file can be placed on the server anywhere to which NetROM has TFTP access.

A given module should be loaded only one time. If you need to reload the module, reset the NetROM unit, then re-execute *loadmodule*.

See Also

di module command, page 4-66

logout

Used to terminate login sessions.

Synopsis

logout

Description

This command terminates a login session. It can be used to exit Telnet login sessions, direct connections on the NetROM control port, or logins on the SLIP port, which are actually a special case of Telnet logins. However, it cannot be used to terminate the NetROM serial console session.

reset

Completely resets NetROM's hardware and software.

Synopsis

```
reset
```

Description

The *reset* command is as effective as power cycling the NetROM unit, but does not affect the contents of emulation memory. This command can be issued from any NetROM terminal session.

stty

Displays or modifies characteristics of NetROM terminal sessions.

Synopsis

stty [-d] { *erase* | *kill* | *werase* | *intr* | *eof* | *alterase*} **setting**
stty [-d] *all*
stty [{ *console*|*target* } [*baud*=**baudrate**] [*stop* = {*1*|*2* }]
    [{ *even* | *odd* | *none* }] [{ *hshake* | *nohshake* }] [{ *xon* | *noxon* }]
stty { *echo* | *noecho* }

Description

The *stty* command allows the NetROM user to customize characteristics of the NetROM terminal session under which the command is invoked. The command can also be used to configure defaults for all subsequent terminal sessions. The optional *-d* flag is used to set or display default characteristics, and simultaneously set control characters for the current session. The *stty* command can also be used to configure both of NetROM's serial ports, the Console Port and the Target Port. Finally, the command can be used to configure NetROM's command interpreter to echo or not echo characters it receives.

All terminal sessions have several control characters associated with them. See "Terminal Control Characters" on page 4-3 for a description of these control characters. The **setting** parameter is of the form "^X"; that is, it is two characters, a carat '^' followed by the alphabetic character itself. One exception is the DELETE key which can be used without a carat "^." If the DELETE key is not mapped as a control character, it will be printed as ^?. The second form of the *stty* command displays terminal control character settings.

The *stty* command can be used to configure either of the NetROM serial ports; the *console* and *target* keywords indicate which port is to be configured. All settings are updated immediately. The baud rate for the port is configured using the **baudrate** parameter. Valid baud rates are 150, 300, 600, 1200,

2400, 4800, 9600, 19200, and 38400. Note that there is no space between the *baud* keyword, the equals sign, and the **baudrate** parameter. The *stop* keyword is used to configure *transmit* stop bits; NetROM's serial ports are always configured for one received stop bit. Parity for a serial port can be set to *odd*, *even*, or *none*. The *hshake* keyword enables RTS/CTS hardware handshaking; if *nohshake* is selected, RTS will be asserted before sending, and CTS will be ignored. NetROM does not support DTR/DSR handshaking; DTR is always true and DSR is ignored. Finally, the *xon* keyword enables XON/XOFF software handshaking, and *noxon* disables it.

If *stty* is invoked with the *noecho* keyword, the terminal session under which it is invoked will stop echoing input keystrokes. This makes the session effectively "half duplex." Input echoing can be re-enabled with the *echo* keyword. Note that command output will not be affected.

Examples

stty eof ^Z

Sets the eof control character for the current terminal session to Control-Z.

stty -d *kill* ^K

Sets the default line-kill character for all terminal sessions to Control-K. This also changes the line-kill character for the current session, but not for any other sessions already running.

stty -d *all*

Displays the current default terminal settings for NetROM terminal sessions.

stty console;baud=4800 *hshake noxon*

Configures NetROM's Console Port to run at 4800 baud with hardware handshaking enabled but XON/XOFF recognition turned off. Other parameters remain as they were before the command was issued.

stty noecho

Disables echoing of keyboard input; this is useful for users establishing TCP connections to the NetROM Control Port, since they may want to handle keyboard input locally, or issue commands directly without echoing them.

See Also

tgtcons command, page 4-28

Environment Variable Commands

NetROM has a special set of pre-defined state variables which are used or referred to frequently by the user. These are referred to as *environment variables* as distinguished from the *generic variables*. Environmental variables are concerned primarily with configuring communications paths between NetROM and the target system, configuring pod groups for emulation, and setting default values for downloading new emulation images. Table 4-8 summarizes the environment variables.

Table 4-8 Environmental variables

Variable	Description
"batchpath"	Sets path on the TFTP file server NetROM uses to search for batch files.
"consolepath"	Sets console communication path between NetROM and the target system. Takes as values: *serial*, *readwrite*, or *readaddr*.
"debugpath"	Sets debug communication path between NetROM and the target system. Takes as values: *serial*, *readwrite*, or *readaddr*.
"debugport"	Sets TCP/UDP port number on which NetROM accepts data from host-based debuggers.
"dprbase"	Sets base address in emulation pod 0 to map dualport RAM.
"filetype"	Sets expected download file format. Supports *binary*, *srecord*, and *intelhex*.
"fillpattern"	Sets byte pattern to fill emulation memory.
"groupaddr"	Sets default pod group's start address.
"groupwrite"	Enables or disables NetROM's external write signal. Takes as values: *readonly* or *readwrite*.
"host"	Sets IP address of the TFTP server used for image and batch downloads.

Table 4-8 Environmental variables (Continued)

Variable	Description
"loadfile"	Sets default file to download into the default pod group.
"loadpath"	Sets default path for downloading "loadfile."
"podgroup"	Sets default pod group.
"podorder"	Sets pod-to-byte mapping of emulation pods in the default pod group.
"romcount"	Sets number of bytes in emulation as part of the default pod group.
"romtype"	Sets ROM type being emulated by the default pod group.
"tgtip" (optional)	Sets target machine's IP address when Virtual Ethernet is on.
"verify"	Specifies whether downloads are verified. Takes as values: *on* or *off*.
"vether" (optional)	Sets Virtual Ethernet on or off.
"wordsize"	Sets size in bits of the ROM word being emulated by default pod group.
"writemode"	Sets mode that configures emulation memory to emulate FLASH ROM or static RAM.

There are two commands which directly manipulate environment variables. These are the *setenv* and the *printenv* commands. The next section describes these two commands and each of the environment variables in detail.

setenv

Allows NetROM users to modify the value of all environment variables.

Synopsis

setenv **variable value**

Description

The *setenv* command allows users to configure NetROM to meet the needs of their development environment. NetROM's environment variables provide a simple and straightforward way to do this, while allowing the user to take advantage of some of NetROM's more advanced features.

The **variable** parameter is the name of the environment variable being set. Table 4-3 summarizes the names and characteristics of NetROM's environment variables. The format of the **value** parameter depends on the variable being set. Consult the documentation for the variable in question for more information on how to specify the value.

printenv

Displays the current values of NetROM's environment variables.

Synopsis

```
printenv
```

Description

The *printenv* command displays the current settings for all environment variables. The variables are summarized in Table 4-3, and discussed in detail in the documentation that follows.

batchpath

The "batchpath" environment variable is specifically designed to facilitate the *batch* command. Some NetROM users may have certain commands sequences that they repeat over and over. It is possible to collect such sequences of commands into "batch files" and have NetROM execute them as a group. Then, using the *batch* command, the NetROM user can cause NetROM to download these files and execute the commands in them one at a time.

The "batchpath" environment variable is the path on the TFTP file server that NetROM should use to search for batch files. It is possible to override this default path if desired. See the *batch* command for details. Note that setting "batchpath" to "*/*" effectively clears it on secure servers, and setting it to "*/tftpboot*" clears it on non-secure servers.

Note

Note to Windows Host Users: Set the batchpath variable to. (dot). This causes NetROM to use the current directory for the search path for batch files on the TFTP file server.

consolepath

The "consolepath" environment variable is used to configure console path communications between NetROM and the target system. Console path communications between the host and NetROM are independent of the path between NetROM and the target, and are not affected by this environment variable.

There are three possible values for the "consolepath" variable. The *serial* value indicates that console path communication with the target should proceed via NetROM's target serial port interface. (Parameters for this port are configured using the *slip* command.) The *readwrite* and *readaddr* values select one of the two emulation memory mailbox protocols. The *readwrite* setting indicates that the target system is capable of writing into its ROM address space, and that message passing will take advantage of this fact. The *readaddr* setting indicates that the target system cannot write ROM space, so the alternate mailbox protocol will be used. These mailbox protocols are described in detail in Chapter 7. Table 4-8 summarizes the possible values for the "consolepath" environment variable.

Table 4-9 Consolepath and Debugpath Environment Variables

Keyword	NetROM-target communication path
serial	NetROM's Target serial port.
readwrite	An emulation memory mailbox protocol which takes advantage of the target's ability to write to its own ROM space.
readaddr	An emulation memory mailbox protocol which uses target-side reads of emulation memory and NetROM-side writes of emulation memory to pass data.

The "consolepath" variable interacts with the "debugpath" variable to some degree. They are somewhat independent, because one may select the serial port communications path and the other a mailbox protocol, and they may both use a mailbox protocol, but they cannot use *different* mailbox

protocols. Table 4-9 summarizes these interactions. In situations where the console path and the debug path between NetROM and the target are the same, the host system side debugger and console will receive both debug and console data. It is the responsibility of the host system to distinguish between them; also see "debugpath" on page 4-106.

Note

Changing the "debugpath" variable will not take immediate effect; the target must be reset with the *tgtreset* command before the target-NetROM communication path will change. This prevents corruption of any current active console sessions.

Table 4-10 Consolepath and Debugpath Interactions

Debugpath	Consolepath		
	Serial	**Readwrite**	**Readaddr**
Serial	Yes	Yes	Yes
Readwrite	Yes	Yes	No
Readaddr	Yes	No	Yes

debugpath

The "debugpath" environment variable is used to configure debug path communications between NetROM and the target system. Debug path communications between the host and NetROM are independent of the path between NetROM and the target, and are not affected by this environment variable.

There are three possible values for the "debugpath" variable. The *serial* value indicates that console path communication with the target should proceed via NetROM's target serial port interface. (Parameters for this port are configured using the *slip* command.) The *readwrite* and *readaddr* values select one of the two emulation memory mailbox protocols. The *readwrite* setting indicates the target system is capable of writing into its ROM address space, and message passing will take advantage of this fact. The *readaddr* setting indicates the target system cannot write ROM space, so the alternate mailbox protocol will be used. These mailbox protocols are described in detail in Chapter 7. Table 4-7 summarizes the possible values for the "debugpath" environment variable.

The "debugpath" variable interacts with the "consolepath" variable to some degree. They are somewhat independent, because one may select the serial port communications path and the other a mailbox protocol, and they may both use a mailbox protocol, but they cannot use *different* mailbox protocols. Table 4-8 summarizes these interactions. In situations where the console path and the debug path between NetROM and the target are the same, the host system side debugger and console will receive both debug and console data. It is the responsibility of the host system to distinguish between them.

Note

Changing the "debugpath" variable will not take immediate effect; the target must be reset with the *tgtreset* command before the target-NetROM communication path will change. This prevents corruption of currently active debug sessions.

---

debugport

The "debugport" environment variable is used to set the TCP/UDP port number on which NetROM listens for data from host-based debuggers. The **portnum** parameter should be in decimal. The default port number is 1235. Note that if a connection-oriented debug session is under way when this command is issued, the new port will not become active until the target system is reset with the *tgtreset* command.

See Also

set udpsrcmode command, page 4-53

dprbase

The "dprbase" environment variable tells NetROM where in emulation pod 0 to map the dualport RAM used to pass messages between NetROM and the target system. The value of this variable is the hexadecimal offset, in bytes, from the start of pod 0 memory. This value is independent of the word size of the pod group containing pod 0. That is, it should be considered the byte offset from the start of the ROM which pod 0 is emulating. By default, the dualport memory is mapped to the last 2048 bytes of memory emulated by pod 0. For example, if pod 0 is emulating a 27c020 ROM, which has 256 Kilobytes, the default "dprbase" is 0x3F800, since this is 2048 (or 0x800) bytes below the end of the ROM, which has 0x40000 bytes.

The "dprbase" variable allows the NetROM engineer to map the communication mailbox area to another part of the ROM, for example to the beginning. The last part of emulation memory was chosen as the default because most ROM users fill their ROMs from beginning to end, not the other way around. If the target system is not going to use dualport memory to pass messages to the target, the value of "dprbase" is unimportant. However, it should not be moved after the pod group containing pod 0 has been downloaded. If dualport RAM will be used to pass messages, it is important that "dprbase" be set so that it does not overlap any of the download image.

Changing the "dprbase" variable does not modify the mapping of dualport RAM immediately. This is to prevent corruption of the current target-NetROM communications path. In order to effect the change in mapping, the target must be reset with the *tgtreset* command.

filetype

The "filetype" environment variable tells NetROM what file format to expect when downloading the default podgroup (which is named by the "podgroup" environment variable). Supported file types and their associated settings are given in Table 4-10. The "filetype" default can be overridden on the command line, when invoking *newimage*, if desired.

NetROM supports extended address records (type 0x04) for the 80386-and-higher processors. These records specify address bits 16 through 31. Old style extended address records (type 0x02), which affect address bits 4 through 19, are also supported.

Table 4-11 Supported NetROM File Types

Value	Meaning
binary	Binary File.
srecord	Motorola S-Record File.
intelhex	Intel hex record file.

fillpattern

The "fillpattern" environment variables allows NetROM users to specify an 8-bit pattern which will be used to fill emulation memory prior to a download. Valid values for this variable are either the keyword *none* or the hexadecimal 8-bit value. Emulation memory can also be filled using the *fill* command. The "fillpattern" value can be overridden on the command line when invoking the *newimage* command. See documentation on *newimage* for details.

groupaddr

The "groupaddr" environment variable gives NetROM the *target's* idea of the start address of the default podgroup (which is named by the "podgroup" environment variable). This value is a 32-bit hexadecimal number, and is intended to allow the NetROM user to examine emulation memory using the same addresses which appear in compiler map files.

groupwrite

The "groupwrite" environment tells NetROM whether or not to allow the target system to perform writes into emulation memory using either the internal or the external write line. A read-only target system cannot write to its own ROM space because its hardware designer did not supply a write line to the ROM sockets. If the target's write cycle is appropriately supported in other respects, the NetROM user may decide to connect the write signal, which is part of NetROM's status signal array, to the target processor's write line.

The "groupwrite" variable *does affect* write cycles which use the write line in the emulation pod. Appropriate values for the "groupwrite" variables are *readonly* and *readwrite*.

host

The "host" environment variable is the IP address of the TFTP server NetROM will use during image and batch file downloads. This address is given in standard dotted-decimal notation. This default address can be overridden on the command line, for example when invoking the *batch* and *newimage* commands.

loadfile

The "loadfile" environment variable is the name of the default file to download into the default pod group (which is named by the "podgroup" environment variable). This should be a simple file name, not a directory path. This file name is concatenated with the "loadpath" environment variable to determine the root-specific path of the default image file. The "loadfile" default can be overridden on the command line, when invoking *newimage*, if desired. Note that setting "loadpath" to "/" effectively clears it on secure servers, and setting it to "/tftpboot" clears it on non-secure servers.

loadpath

The "loadpath" environment variable is the default directory path NetROM will use when downloading image files into the default pod group (which is named by the "podgroup" environment variable). This path may or may not be "root-specific"; that is, it may or may not begin with a '/'. Most TFTP servers will treat non-root-specific paths as being based out of the */tftpboot* directory. The "loadpath" default can be overridden on the command line, when invoking *newimage*, if desired.

Note

Note to Windows Host Users: Set the loadpath variable to "." (dot). This causes NetROM to use the current directory as the default directory for the file being transferred from the TFTP default directory on the host.

podgroup

The "podgroup" environment variable indicates which pod group should be considered the default podgroup for various commands that affect podgroups. These commands include *newimage* and *di podmem,* among others. In addition, the default podgroup is the one which is acted upon by the environment variables affecting .ipod group:configuration configuration; and downloading. The value of this variable is the pod group number which should be selected as the default.

podorder

The "podorder" environment variable (illustrated in Figure 4-1) maps emulation pods within the default pod group (which is named by the "podgroup" environment variable) to ROM sockets being emulated by that pod group. For example, if pods 0 and 1 are members of pod group 0 which emulates a 16-bit-word target ROM space, it may be desirable for pod 0 to emulate ROM 0 while pod 1 emulates ROM 1, or vice versa.

The pod order is specified using pod numbers separated by colons (':') and dashes ('-') to indicate "parallel" or "serial" pods respectively. Parallel pods occur within single ROM words and serial pods occur within consecutive words. For example, the notation "0:1" indicates that pods 0 and 1 work together to emulate a 16-bit word, in which pod 0 emulates byte 0 and pod 1 emulates byte 1. The notation "0-1" indicates that pods 0 and 1 work together to emulate an 8-bit word, where pod 0 emulates the lower-addressed words and pod 1 emulates the higher-addressed words. This second notation indicates an emulated space where words are half as wide as the first notation, but in which there are twice as many words. Both serial and parallel pods may occur in podorder notation; for example, "0:1-2:3" indicates not only that pods 0 and 1 emulate a 16-bit word, as do pods 2 and 3, but also that the words emulated by pods 0 and 1 are lower-addressed than the words emulated by pods 2 and 3.

The "podorder" variable is related to the "romcount" and the "wordsize" variables. That is, the "podorder" contains within it the combined information of the "romcount" and "wordsize" variables. The "podorder" variable will override both the "romcount" and "wordsize" variables. However, the "romcount" and "wordsize" variables are probably more intuitive to use. Without the "podorder" variable, the order of pods within the default pod group is always the same. Figure 4-2 summarizes the interactions of the "romcount" and "wordsize" variables, and gives the default values of the "podorder" variable for each case. The "podorder" variable can be used to explicitly set the mapping between pods and ROM sockets if desired.

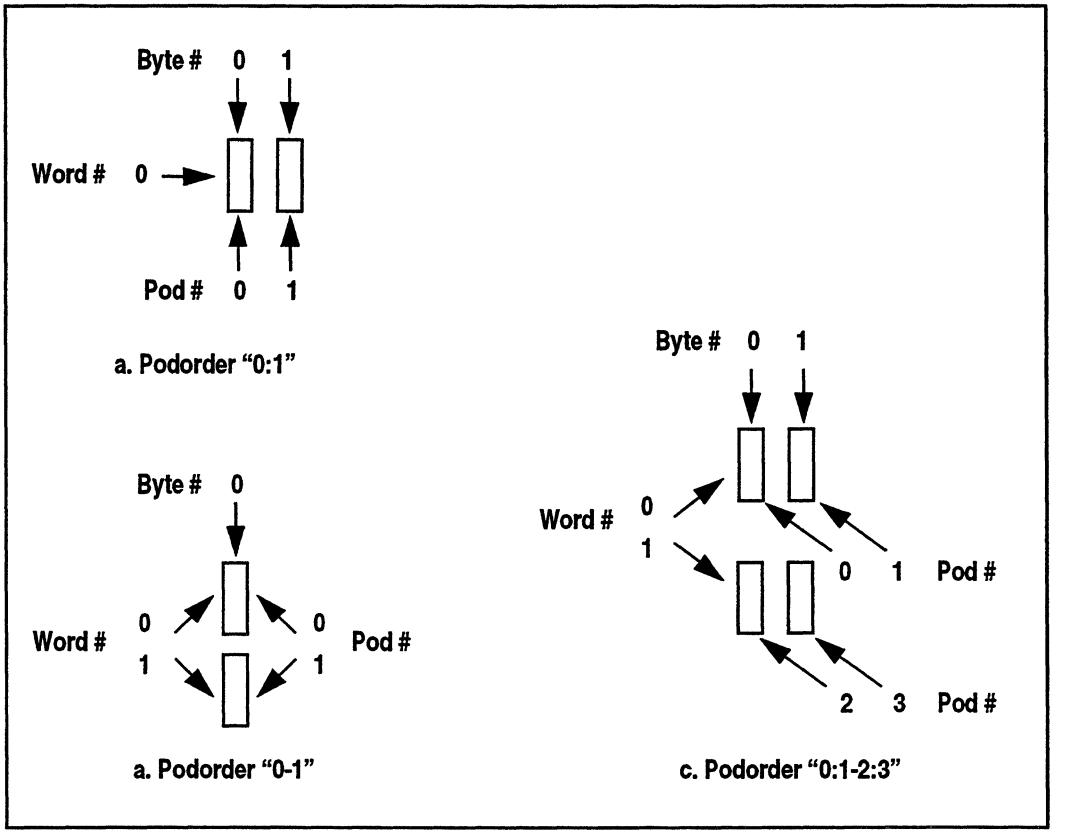


Figure 4-1 Podorder Examples

romcount

The "romcount" environment variable is concerned with the emulation of the default pod group (which is named by the "podgroup" environment variable). The "romcount" variable is related to the "podorder" and the "wordsize" variables. The "podorder" contains within it the combined information of the "romcount" and "wordsize" variables. The "podorder" variable will override both the "romcount" and "wordsize" variables. However, the "romcount" and "wordsize" variables are probably more intuitive to use. Without the "podorder" variable, the order of pods within the default pod group is always the same. Figure 4-2 summarizes the interactions of the "romcount" and "wordsize" variables, and gives the default values of the "podorder" variable for each case.

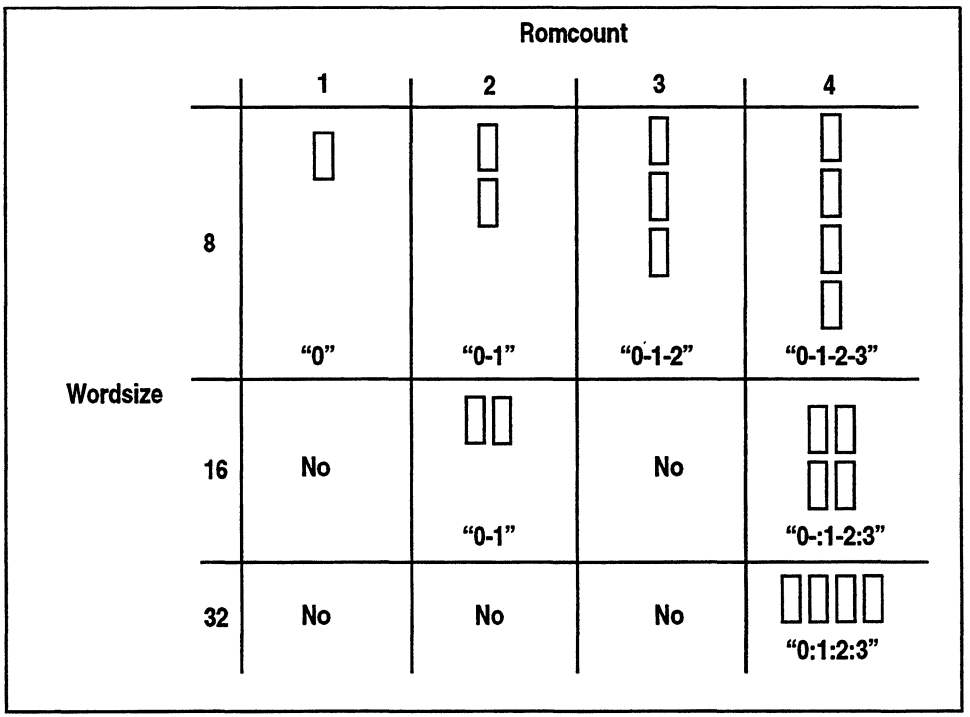


Figure 4-2 Podorder/Romcount/Wordsize Interactions

romtype

The "romtype" environment variable specifies the type of ROM that NetROM will emulate. Table 4-12 gives the possible ROM types, their corresponding ROM sizes, and the acceptable variable name to use when setting the romtype environment variable. The variable names are case sensitive.

Note that some of these ROMs are 16 bits wide. Such devices, and the 4 Megabit 27c040, require an "active pod" that combines two of NetROM's emulation pods into one DIP or PLCC plug. Note that many of the type names are aliases for one another.

If your ROM part is not list, use one of the generic types. NetROM does not distinguish between ROM types that are the same size.

Table 4-12 ROM Types and Sizes

ROM type	Size	ROM type variable name	Attribute
27c256, 28f256	32K x 8	27c256, 28f256 32k_by_8	
27c512, 28f512	64K x 8	27c512, 28f512 64k_by_8	
27c010, 28f010, 28f001b, 27c100	128K x 8	27c010, 28f010, 28f001b, 27c100 128k_by_8	
27c020, 28f020	256K x 8	27c020, 28f020 256k_by_8	
27c040, 29fo40	512K x 8	27c040, 29fo40 512k_by_8	read-only
27c1024, 27c210	64K x 16	27c1024, 27c210 64k_by_16	

Table 4-12 ROM Types and Sizes (Continued)

ROM type	Size	ROM type variable name	Attribute
27c2048, 27c220	128K x 16	27c2048, 27c220 128k_by_16	
27c4096, 27c400	256K x 16	27c4096, 27c400 256k_by_16	read-only
27c400	256K x 16 or 512K x 8	27c400, 256k_by_16 512k_by_8	read-only

Read-only ROM types cannot be written directly by the target, even using the external write line. To write to these ROMs, use the read-address protocol to request that NetROM modify emulation memory. This can be done even if neither the debug path nor the console path use the read-address protocol. All other ROM types can be written if (a) the pod group being emulated is read-write, and (b) the write signal in the pod or the external write line is asserted with normal write cycle timing. If the ROM being emulated has both its write signal and its output enable asserted, NetROM assumes that the cycle is a read cycle, aborts the write without modifying emulation memory, and drives the data bus with the contents of the ROM at the specified address.

See Also

set raconfig command, page 4-46
set rawrites command, page 4-48

tgtip (optional)

The "tgtip" environment variable specifies the target machine's IP address when Virtual Ethernet is turned on (see "wordsize" on page 4-126). This IP address determines which packets to send to the target. Enter this address in dotted decimal form; e.g., 192.3.4.5. (Note: Virtual Ethernet is an optional feature of NetROM, it is enabled when the licensed Virtual Ethernet RAM module is loaded.)

verify

The "verify" environment variable affects NetROM's behavior as it downloads pod groups. Valid values for this variable are *on* and *off*. When "verify" is *on*, NetROM calculates the checksum of hex records and compares the sum with the value given in the record. The "verify" variable has no effect on binary file downloads. Since verifying records involves additional arithmetic steps, it will tend to slow the download process slightly.

vether (optional)

The "vether" environment variable turns Virtual Ethernet on and off. When the corresponding driver is running on the target system, Virtual Ethernet enables NetROM to act as an Ethernet interface for the target. Virtual Ethernet filters incoming packets and sends those addressed to the target over the dualport RAM interface to the target. Packets from the target are transmitted on the Ethernet. (Note: Virtual Ethernet is an optional feature of NetROM, which is enabled when the Virtual Ethernet RAM module is loaded.)

writemode

Synopsis

```
setenv writemode {flash | static}
```

Description

The "writemode" environment variable configures the type of device which will be emulated when writing to emulation memory from the target. This is important if both OE and WR are asserted at the same time. With flash ROM emulation, asserting OE and WR causes a READ cycle. With static RAM emulation, asserting OE and WR causes a WRITE cycle. The default is flash ROM.

Note

This optional feature requires a factory hardware modification to switch modes. The standard product emulates flash ROM.

wordsize

The "wordsize" environment variable indicates the width in bits of the words emulated by the default pod group (which is named by the "podgroup" environment variable). Valid word sizes are 8, 16, and 32 bits.

The "wordsize" variable is related to the "romcount" and the "podgroup" variables. The "podorder" contains within it the combined information of the "romcount" and "wordsize" variables. The "podorder" variable will override both the "romcount" and "wordsize" variables. However, the "romcount" and "wordsize" variables are probably more intuitive to use. Without the "podorder" variable, the order of pods within the default pod group is always the same. Figure 4-2 summarizes the interactions of the "romcount" and "wordsize" variables, and gives the default values of the "podorder" variable for each case.

Chapter 5
Debugger Support

NetROM provides support for embedded systems developers using remote debuggers. Remote debuggers are software systems which run both on the host system and on the target. Most remote debuggers use RS-232 serial links to connect their target and host sides. NetROM removes the need for serial links; data packets destined for the target system's half of the debugger can be sent to NetROM over Ethernet and forwarded from NetROM to the target along the configured debug path. Similarly, data from the target will be forwarded by NetROM to the host.

The NetROM approach has several advantages. First, it does not require that the host system running the debugger user interface have a serial port. Second, it allows the system side of the debugger to use system calls which interface to a TCP/IP network; this is often simpler and more portable than writing software to program a serial link. Third, NetROM can be used to debug target systems which do not have a serial port; NetROM insulates the host side of the debugger from the details of communicating with the target.

NetROM Debug Paths

NetROM provides three choices for the debug path to the target. This allows the NetROM user to choose the option that best suits the requirements of the development environment and the target system. The first option is the serial debug path. This works well in environments which currently use serial links to communicate with the target. NetROM can be used in these environments with no target-side modification at all; the target sends and receives debugger packets on its serial port.

Targets which do not have serial ports can be separated into two categories: those which can write to their ROM space and those which cannot. NetROM can pass messages to both types of target systems using portions of emulation memory as mailboxes. It uses two different mailbox protocols, one for each type of target system. Details of these protocols are given in Chapter 7. In sum, targets able to write emulation memory can pass data to NetROM more quickly than those which cannot. However, the protocol for targets which cannot write emulation memory will work with *all* target systems.

Passing Data Across the Debug Path

The mechanism for host side debuggers to pass data to and from the target system is quite simple. NetROM has a "daemon" process, called "debugpathd," which listens on a specific TCP port, the Debug Data Port. The port number of the Debug Data Port is given in Appendix C. In order to send data to the target, the host side of the debugger needs only to establish a TCP connection to the Debug Data Port. This can be done in a straightforward way on most system using well-defined system calls. Data for the target can be sent on this connection and data from the target can be received on it.

NetROM's "debugpath" environment variable configures the NetROM-to-target communication path. The default path uses NetROM's Target Serial Port. The path from NetROM to the host system is independent of the NetROM-to-target path. The target must be reset with the *tgtreset* command before changes to the debugpath will take affect, even in the NetROM startup file.

The Debug Control Port

In addition to simply providing a facility for passing data between the host system and the target sides of a debugger, NetROM provides a mechanism for debuggers to directly control the target. This is done through the Debug Control Port. The Debug Control Port is a TCP port (whose number is given in Appendix C), which is monitored by the "debugctld" process. The Debug Control Port allows the host side of the debugger to communicate with NetROM, and allows it to perform many of the functions which are available on the NetROM command line. These functions include, among others, resetting the target, examining and/or writing emulation memory, and downloading a new image.

Currently the Debug Control Port simply accepts ASCII text in the form of NetROM command line commands. There is no mechanism for machine-readable feedback.

Debug Control Functions

These include resetting the target, displaying and setting emulation memory, and downloading new images. The current implementation of the Debug Control Port is that it simply provides a command-line interpreter, similar to the NetROM Control Port. Although this mechanism is likely to change in the near future, current implementations can treat the Debug Control Port connection as if it were a NetROM Control Port connection and achieve results in the short term.

Chapter 6
Alternate NetROM Interfaces

NetROM can be used in environments which do not support TELNET or TFTP. These protocols are essential for "normal operation" because they make it easy for users to "log in" to NetROM or to download files to emulation memory using off-the-shelf software. However, NetROM also provides facilities that allow users to perform these same functions with software they write themselves.

Non-TELNET Terminal Sessions

The NetROM "netromd" process listens on the NetROM Console Port. This is a TCP port whose number is given in Appendix C. In order to obtain a command-line interface to NetROM, it is merely necessary to connect to this socket using standard system calls which interface to TCP. Such a program could be part of a simple terminal emulator, which monitors both its local keyboard and the NetROM connection for activity, or it could be part of a more complex program which wants to be able to make NetROM perform various actions. An example of the latter program might be an X-Windows interface to NetROM which offers a point-and-click interface for commonly used functions.

Unlike TELNET connections, the NetROM Console Port connection is half-duplex, so characters NetROM receives will not be echoed. This can be configured using the *stty* command; see the description of *stty* for details. To exit the connection, simply close the socket.

Non-TFTP File Downloads

It is possible to "send" a file to NetROM for download into emulation memory, rather than have NetROM "request" the file with TFTP. This facility might be useful for the host side of target system debuggers, if they wish to provide debugger extensions which download new emulation images.

Downloading files without TFTP is done using a TCP socket. There are three phases of the download. The first is initiating the download. This is more complicated than simply connecting to a socket, because it is necessary to specify which pod group will be downloaded. The mechanism for initiating a download is to send an ASCII string similar to a command-line command. The string consists of a *download* command, which is only available on the NetROM Console Port. The syntax for the *download* command is identical to that of the *newimage* command, except that the *host=* keyword and the **filename** parameter are not recognized. If no connection is established on the NetROM Download Port after a few seconds, the download will be aborted.

The second phase is opening a TCP connection to the NetROM Download Port. The number for this port is given in Appendix C. To send the file, simply write its bytes to the connection. The third phase of the download is termination. Simply close the TCP connection to signal NetROM that the download is complete.

Uploading Emulation Memory

It is possible to have NetROM "send" the contents of emulation memory to the host system on a TCP connection. This might be useful for verifying the contents of emulation memory. As with downloading files, there are three phases of the upload process. The first is initiating the upload. This is more complicated than simply connecting to a socket, because it is necessary to specify which pod group: will be uploaded. The syntax for the *upload* command is "*upload* **podgroup**". This command is only available on the NetROM Console Port. The command is invoked with the number of the pod group to be uploaded, and activates the NetROM Upload Port. The second phase is opening a TCP connection to the NetROM Upload Port. The number for this port is given in Appendix C. The contents of emulation memory can be read as binary data from the Upload port. The third phase is termination; upon reaching the end of emulation memory for that group, NetROM will terminate the connection. If no connection is established on the NetROM Upload Port for a few seconds after the *upload* command is issued, the upload will be aborted.

Chapter 7
Emulation Memory Mailbox Protocols

NetROM provides two non-RS-232 protocols for communicating with target systems. Both of these pass messages in emulation memory. Potentially they can be very fast, since they are essentially memory-to-memory transfers between NetROM and the target system, and since the link between NetROM and the host system is a high-speed LAN. NetROM provides two protocols, because some targets, which cannot write to their own ROM space, need to use an alternate mechanism to pass messages to NetROM. This chapter describes in detail the protocol used between NetROM and the target system. Sample target-side implementations of these protocols are available with NetROM free of charge.

Sharing Emulation Memory

In order to explain the implementation of target-NetROM protocols in shared memory, it is necessary to describe some aspects of ROMs. ROM devices do not have an "output valid" signal. Instead, ROM accesses are, in some sense, asynchronous to the system clock. The target asserts an address on the ROMs address lines, waits a certain number of system clock cycles, then latches in the data on the ROM output lines. This sequence of events constitutes a ROM *memory cycle*. Figure 7-1 shows how this works. This can be considered asynchronous, because ROMs do not use the system clock to latch asserted addresses or outgoing data.

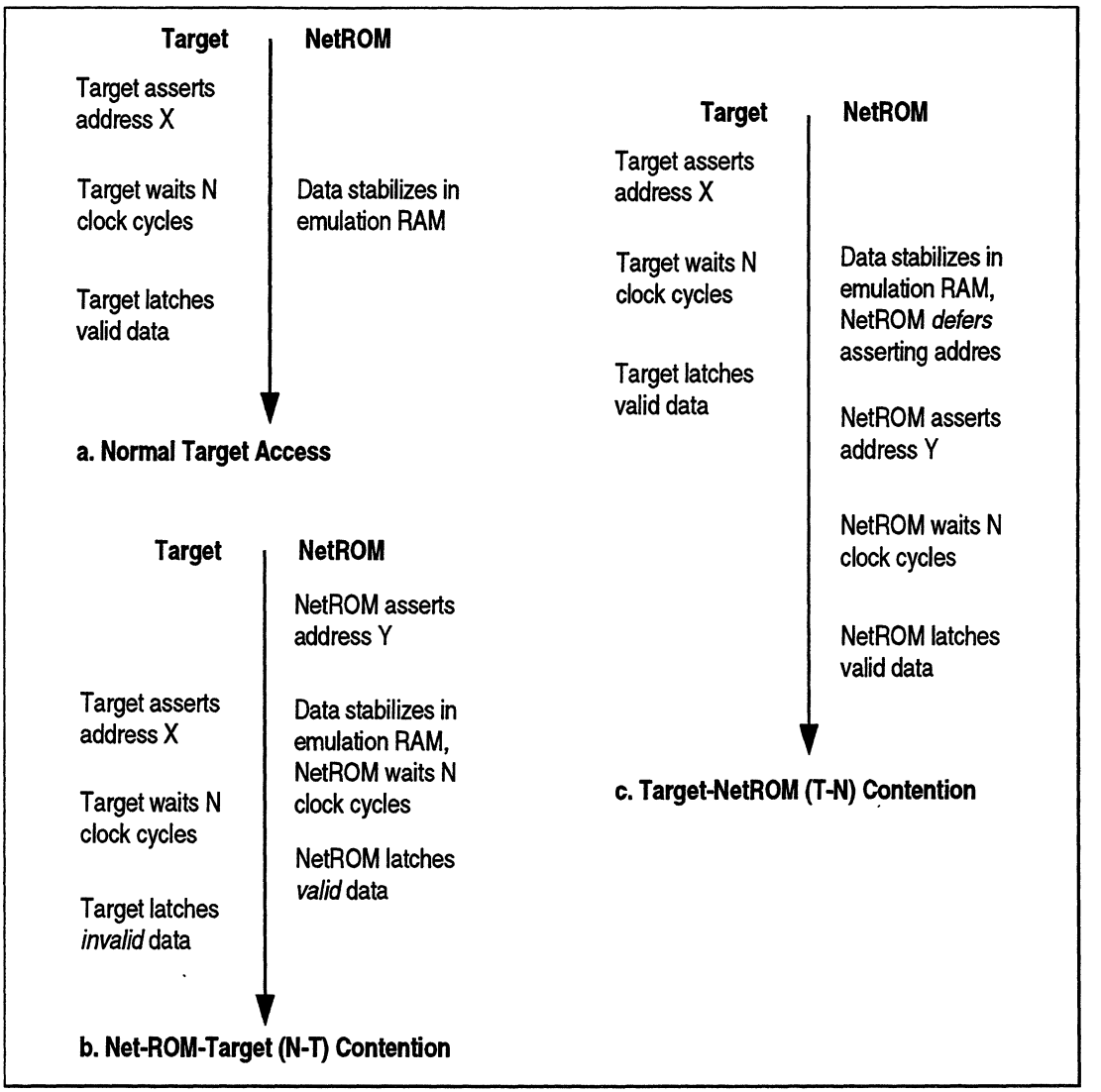


Figure 7-1 Memory Access and Contention Cycles

Memory Contention Issues

The timed method of accessing ROMs has unfortunate consequences for passing messages between the target and NetROM in emulation memory. If only one party; i.e., NetROM or the target, accesses an emulation pod at a time, everything is fine. In normal operation, only the target will access emulation memory. However, when passing messages it is necessary for both parties to read and possibly write to a "ROM." This leads to two forms of contention.

The first form of contention occurs when NetROM accesses an emulation pod, and the target attempts to access the same pod before NetROM's access is completed. There is no way for NetROM's memory hardware to tell the target that the pod is busy and that the target should expect a delay in receiving its data. Instead, the target will wait its prescribed number of clock cycles and latch in the wrong data. This situation is shown in Figure 7-1. We shall refer to this as *N-T contention*, since the NetROM, then the target, attempt to access the same memory during a single memory cycle.

The second form of contention occurs when the target accesses an emulation pod, and NetROM attempts to access the same pod before the target's cycle is competed. Unlike the target side, NetROM's hardware is capable of "holding off" the NetROM ROM cycle until the target's is finished. Unfortunately, if the target is very busy accessing that particular pod, NetROM's processor may be held off indefinitely. This is an undesirable occurrence, and may cause bus errors on the NetROM side. We shall refer to this form of contention as *T-N contention*, since the target, then NetROM, attempt to access the same memory during a single memory cycle (see Figure 7-1). Note that although the target will get correct data during its access cycle, there is no way to guarantee that it will "beat" NetROM to the memory. Therefore, T-N contention is just as unreliable and undesirable as N-T contention.

Dualport Emulation Memory

To address the problems of T-N contention, NetROM provides memory in emulation pod 0 which is "special." This memory is *dual ported*, which means that it is capable of supporting, not one, but *two* simultaneous access cycles. That is, when the target asserts an address and begins its waiting period, the NetROM is able to assert a different address and begin *its* waiting period, and both parties will receive correct data. This is different from normal ROM or RAM, which can only supply data for one address at a time.

However, even dualport RAM does not completely solve the problems which occur when both the target and NetROM access the *same* memory location. When both parties access the same address, contention again occurs. However, the bad effects of the contention are much reduced. Dualport RAM supplies an "address busy" signal which will cause the NetROM hardware to back off during target cycles. The access will be completed as soon as the target is done with its cycle; thus, T-N contention is averted entirely. However, since this signal is ignored by the target, the target may get corrupted data if it begins its cycle after NetROM does, so there is still a potential for N-T contention. This problem is particularly acute when the target attempts to write data to dualport RAM; if it begins its write cycle after NetROM begins a read cycle, the data which it attempts to assert will be lost.

To solve the problems posed by N-T contention, which is an unavoidable consequence of the way in which ROMs work, NetROM uses a software protocol. This protocol uses messages written into dualport RAM, and comes in two forms, one for targets which can write their ROM space, and one for targets which cannot. Both protocols essentially keep the NetROM from accessing any address at the same time as the target more than once. The address at which potential contention can occur is well defined, and the target will read garbled data *at most once*. This is because N-T contention can occur for only one cycle. These two protocols are described in detail in subsequent sections.

NetROM's 2048 bytes of dualport RAM is located in emulation pod 0. Its location within the pod is configurable. This is because the dualport memory is physically separate from pod 0's emulation RAM and can be substituted for any 2K portion of pod 0. That is, the NetROM user can select the address at which dualport memory will start. Subsequent accesses to the dualport address range will behave exactly the same as accesses of normal emulation memory, except that contention problems are reduced as described above. Note that if dualport memory is moved, its contents will move with it. This means that if dualport memory overlaps part of, say, the target system's executable image, and dualport RAM's address is re-mapped, that part of the target system's image will appear to move. This is shown in Figure 7-2. The lesson to this is twofold: don't allow dualport RAM to overlap image data if possible, and, if an overlap is necessary, re-download the image after moving dualport RAM. Note that the address of dualport RAM is affected by NetROM resets, so setting its location should be part of the NetROM startup file. The address defaults to the highest-addressed 2Kbytes emulated by pod 0.

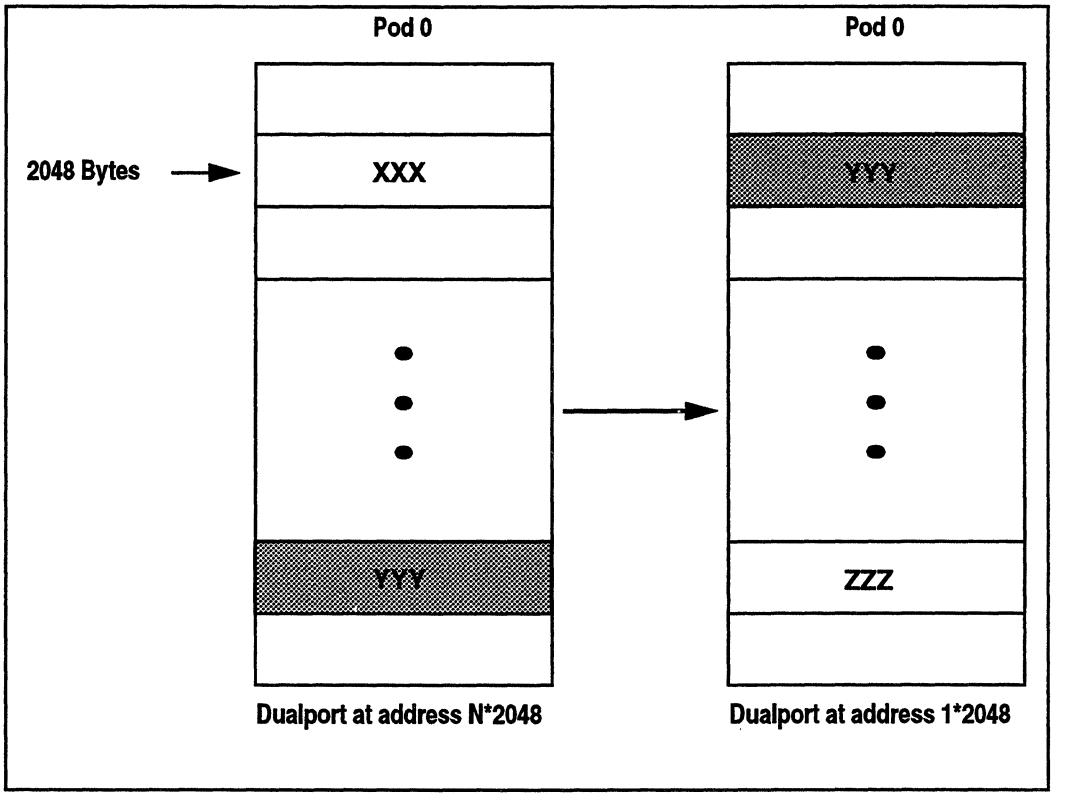


Figure 7-2 Re-mapping Dualport RAM

The Dualport Message Structure

Messages written to dualport memory employ a common structure, whether the target system is write-capable or not. Naturally, read-only targets cannot write messages; they use the read-address memory protocol described below.

Dualport message structures (DMSs) have three parts, a *flags* part, a *size* part, and a *data* part. Each of these parts is of a fixed size, as described in Figure 7-3. Essentially, the party writing the message to dualport RAM writes the data, then the

size, then the flags. Most of dualport RAM is used as arrays of these structures, one array written by NetROM and possibly another array written by the target. A more complete description of the arrays is given for the particular protocol used by the target.

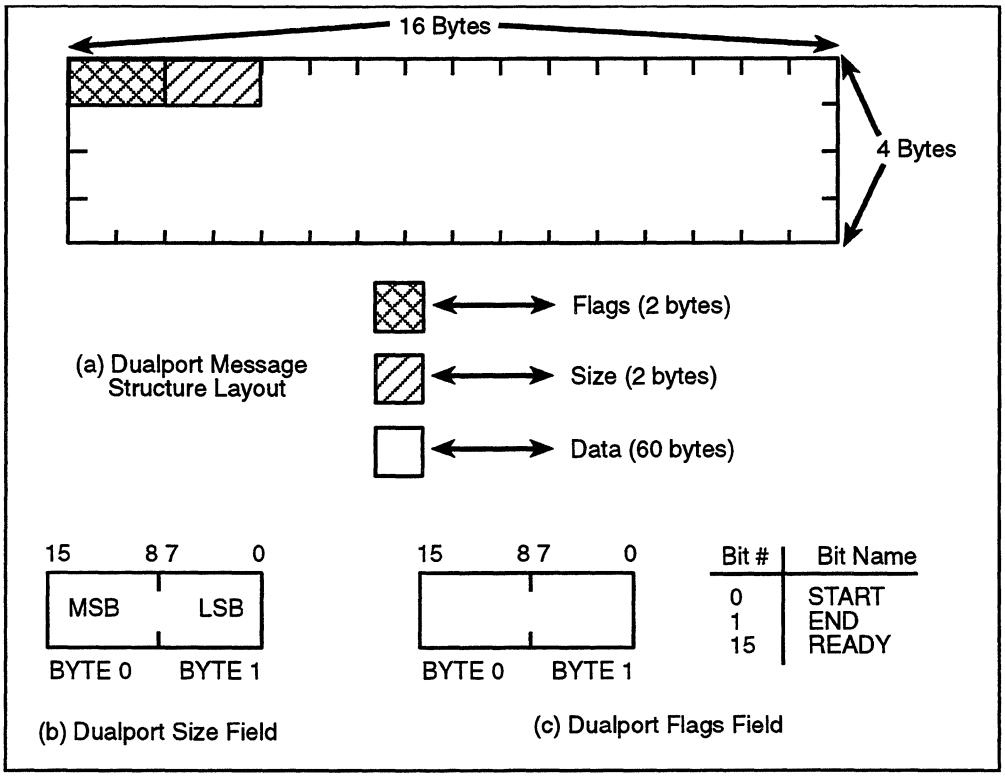


Figure 7-3 Dualport Message Structure

The Size field is interpreted as a big-endian value; that is, the lower-addressed byte contains the more significant bits of the address. The layout of the Size field is given in Figure 7-3.

The Flags field is used to indicate when the message is complete and ready to be processed by the recipient, whether

the particular message is the start, end, or both, of a larger message, and whether the particular message is the last of a given array. The layout of the Flags field is given in Figure 7-3. Note that the byte containing the READY bit should be written last, after all other bytes of the message are valid.

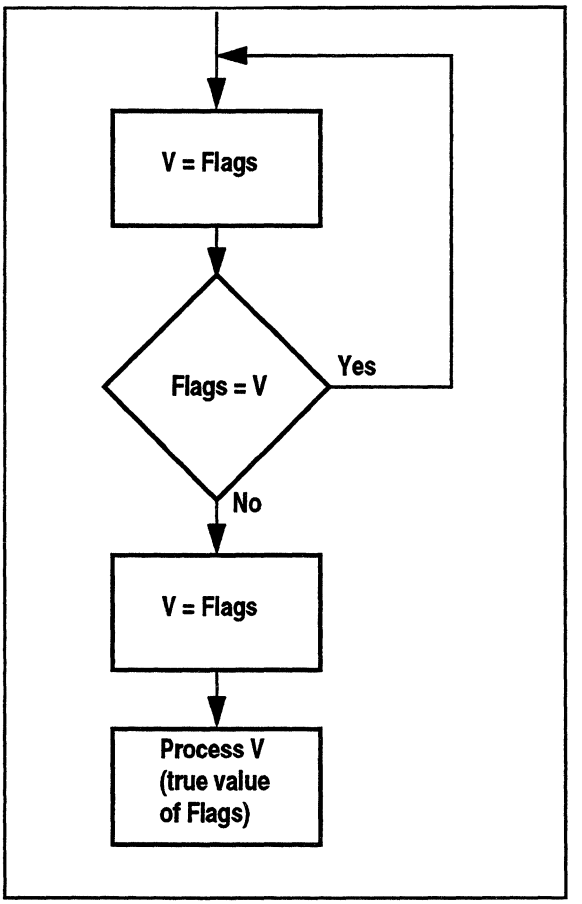


Figure 7-4 Target Validating Dualport Flags

While the target system is waiting for a message to arrive from NetROM, it will poll the Flags field of the next expected message. Note that, due to N-T contention, the target may detect a change in the value of the Flags field, but it cannot be

sure that it has read the *correct* value until it has read it twice without seeing a change. Both of the dualport protocols guarantee that NetROM will only write the Flags field once, so once the target sees a "stable" value, it knows it is valid. *Reading other message fields twice is unnecessary*, since the protocol is set up to only allow potential contention to occur on the Flags field. The flowchart in Figure 7-4 depicts the target system reading and verifying the Flags field of a dualport message structure.

Read-address Memory

As has been described above, it is not a good practice to have both NetROM and the target constantly accessing ("polling") the same area of memory. More particularly, if NetROM is constantly polling an address waiting for the target to write something, N-T contention may corrupt the target's written data, and may also prevent the target from reading back what it wrote to verify its correctness. Thus target-to-NetROM communication using dualport RAM is interrupt driven. The mechanism for this is *read-address memory*.

NetROM's read-address memory is another special area of pod 0; 256 bytes in size, it is separate from dualport memory, but part or all of it may overlap dualport memory. When the target *reads* from this address range, the offset of the address read is latched by NetROM's memory hardware and an interrupt is generated to NetROM's processor. This enables two things to happen. First, it provides the target with a means to inform NetROM of events via interrupts. An example of such an event might be the target writing a message into dualport RAM. Note that using this mechanism to notify NetROM of messages waiting to be read allows NetROM to avoid polling "flag" locations and possibly garbling the target's attempt to write them. The other aspect of read-address memory is that the 8-bit offset latched in by NetROM's memory hardware can be interpreted as an 8-bit data value. Thus, targets which cannot write to their own ROM space can use read-address memory to pass data to NetROM.

Read-address memory is similar to dualport memory because its address is relocatable. Current implementations of the dualport protocol locate the read-address memory at the low-addressed end of dualport RAM, so that when dualport RAM moves, so does the read-address RAM. Read-address dualport RAM can be written by hosts capable of doing so, but any access, even a write to this range of memory will cause address latching and an interrupt to NetROM. To prevent this from causing confusion for write-capable targets, NetROM is able to selectively enable portions of the read-address RAM, and in fact enables only the first 8 bytes of it for targets using the read-write dualport protocol. When the target system is not using a dualport protocol the read-address RAM's interrupt capability is disabled entirely. Other than causing interrupts to NetROM's processor, read-address memory is completely normal dualport RAM.

NetROM will receive only one interrupt from the target, no matter how many times the target reads read-address memory, and that only the first address read will be latched. Only after NetROM has serviced the interrupt and read the latched value will a new interrupt and associated value be enabled. NetROM requires a software protocol to make practical use of the read-address memory. Essentially, NetROM will set a flag in dualport RAM indicating that it is ready to receive read-address interrupts. The target may then read some address within the enabled read-address range. NetROM will service the interrupt this causes, read the latched 8-bit offset for the start of read-address memory, and increment an "acknowledge" byte in another part of dualport RAM. The target polls the "acknowledge" address after it reads looking for the incremented value. Note that, due to N-T contention, the target needs to read the "acknowledge" value two or more times to verify the value that was written. This is the same process as that depicted in the flowchart in Figure 7-4, which shows the target reading and verifying the Flags field of a dualport message structure. The variations on this, the *read-address protocol*, are given for each variety of the dualport mailbox protocols.

NetROM supports target systems whose memory interface hardware is always "burst reads" from emulation memory. In older implementations of the read-address protocol, this caused problems on 32-bit systems which ran from a single 8-bit ROM. The target processor's memory interface hardware on such systems may read 4 consecutive bytes to assemble a single 32-bit word to present to the processor. If the interface *always* reads 32 bits, even on a byte access, then the effective size of the read-address area of dualport memory is 256/4 = 64. To accommodate such target systems, NetROM can be configured to expect burst reads, and the sample implementation of the target-side interface driver has been changed to take burst reads into account. Consult documentation on the *set raconfig* command for more information.

Note

NetROM version 1.2.6 will not interoperate with targets running the previous version of the read-address protocol.

Read-write Targets

The dualport *readwrite protocol* may only be used by target systems which are capable of writing to their own ROM space. Such systems include those which use FLASH ROMs capable of being reprogrammed by the target system's processor with new images. The readwrite protocol allows the target system and NetROM to exchange data in chunks the size of the DMS data field (described in Figure 7-3). This amounts to a memory-to-memory transfer between the target system and NetROM' s processor.

Figure 7-5 shows the layout of dualport memory when the readwrite protocol is used. Remember that dualport memory may be mapped anywhere within pod 0, and that its default location is at the top 2K of the ROM emulated by pod 0. Note that dualport RAM is divided into two arrays of message

structures; one array is written by the target, and the other by NetROM. Note also that the first 64 bytes of dualport memory is used for a configuration/status structure, and that read-address memory overlaps its first 8 bytes.

The configuration/status structure has only three active one-byte fields; the remaining bytes of the structure are reserved and should not be accessed. The TXA and RXA bytes are set to one by NetROM when the transmit and receive arrays (or *channels*) become active at the start of a session. The target should verify that they are set before performing any activity in dualport RAM. However, once set, they will remain set until the target system is reset with the *tgtreset* command. The address of the MRI, or Message Ready Indicator, is used to indicate to NetROM that the target has written a message. Note that it is part of the Interrupt Area (and the only part of the Interrupt Area shared with the Read-write RAM), so reading its address sends an interrupt to NetROM.

Read-write Target-to-NetROM Message

The flowchart in Figure 7-6 depicts the target system sending a message to NetROM, and Figure 7-3 describes the message structure. Sending a message has three stages. The first is obtaining the next free message structure. A free message structure is one in which the READY bit is not set. If this bit were set in the message, it would mean that NetROM had not yet processed the message, which must have been written previously. The target system should wait, or perform other processing, until the next DMS structure becomes free. The second stage is writing to the message data and length fields. This is done in a straight forward way. The third and final stage is notifying NetROM that the message is ready. This entails setting the READY bit in the message's Flags field, and reading the MRI byte. (The actual data at the MRI address is meaningless.) When NetROM has received the interrupt and processed the message, it will clear the READY bit in the Flags field, and the target may reuse the structure.

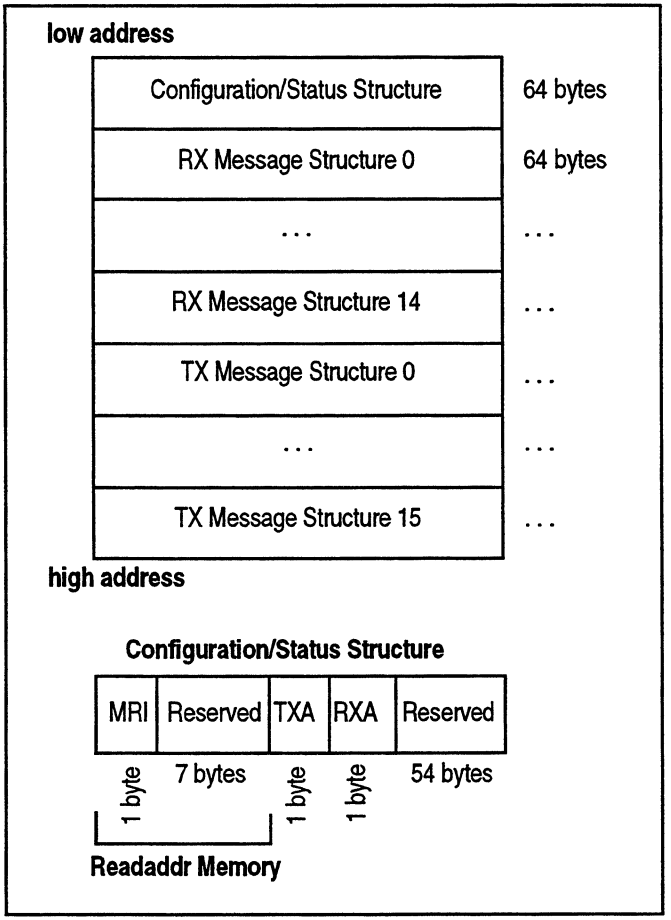


Figure 7-5 Dualport Read-write RAM

Read-write NetROM-to-target Message

The flowchart in Figure 7-6 depicts the target system receiving a message from NetROM. Like sending a message, receiving a message has three stages. The first is detecting a new message. Since NetROM cannot send an interrupt to the target system, the target must "poll" the Flags field of the next message it expects to get. Due to possible N-T contention, it must verify the Flags value by reading it twice. The second stage is copying

the data out of the message structure and processing it. The third stage is clearing the READY bit in the Flags field, which returns the message structure to NetROM for reuse.

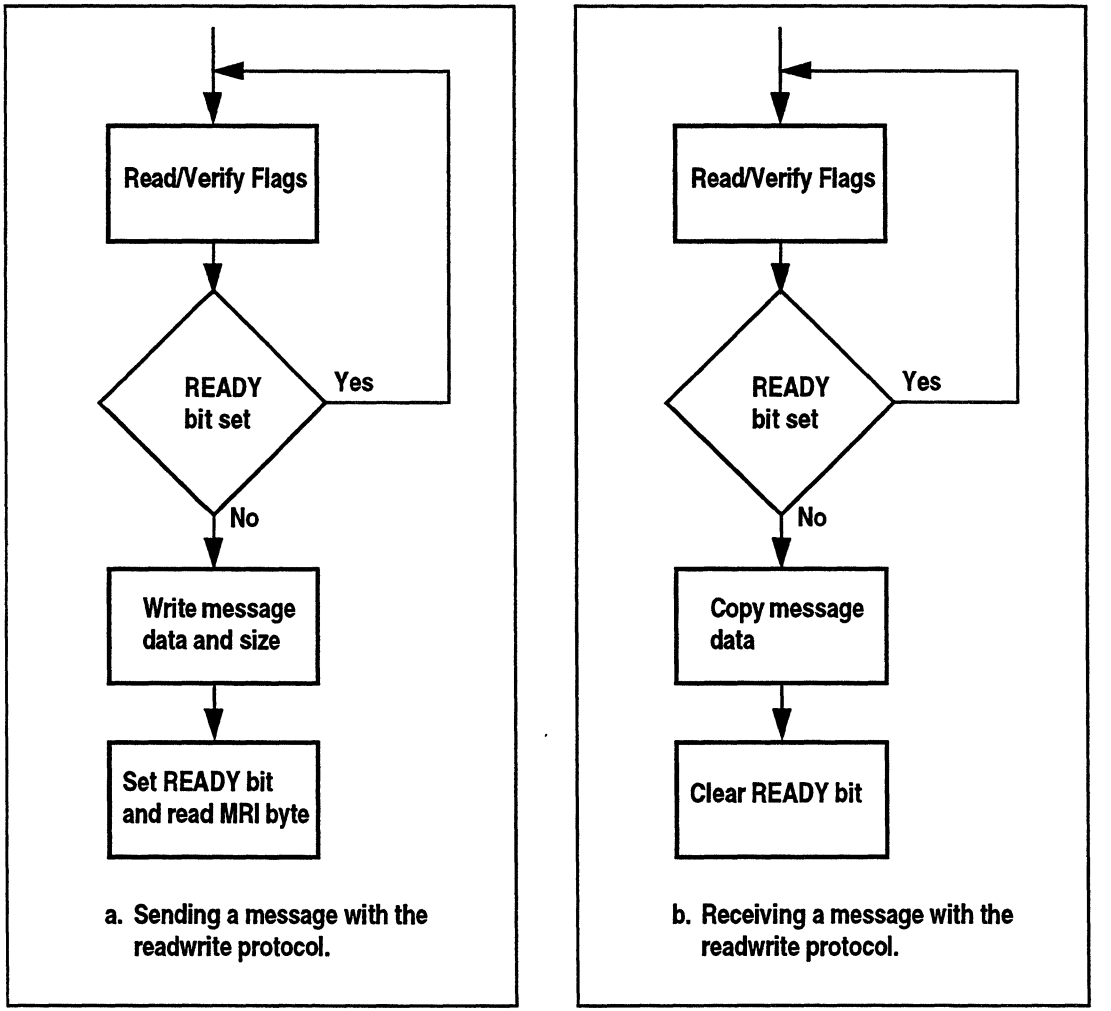


Figure 7-6 Dualport Protocol for Read-write Target Systems

Read-only Targets

The dualport mailbox *readadd protocolr* may be used by any target system. It is particularly useful for targets which have no serial port, and which cannot write to their own ROM space. NetROM sends messages to the target using dualport message structures, and the target sends acknowledgments and data to NetROM via the read-address memory mechanism. Figure 7-7 shows the layout of dualport memory when the *readaddr protocol* is in use.

Remember, dualport memory may be mapped anywhere within pod 0, and that its default location is at the top 2K of the ROM emulated by pod 0. This protocol places the entire 256-byte read-address memory at the start of dualport RAM, and divides it into two sections: *data section* and *control section*.

The data section of read-address memory is used by the target to pass data bytes to NetROM. That is, if a read-address interrupt is determined to have been caused by a target access of this area, the 8-bit value obtained from the read is interpreted as "frame data" destined for the host system, not the NetROM unit.

Note that the layout of the data area is not rigidly defined. This is because some target systems do not have ROM interface hardware that is capable of making a single access to ROM space, even when attempting to read only a single byte. For example, some targets might have 16- or 32-bit processors reading their opcodes from a single 8-bit ROM. Hardware designers for these systems may decide not to support single 8-bit accesses to the ROM space, but instead to burst read 2 or 4 bytes even when the processor requests only 1. Other systems may include ROM space in cached memory and perform burst reads from ROM for that reason. If possible, burst refills of cache from the read-address data area should be disabled since they reduce the performance of the read-address protocol.

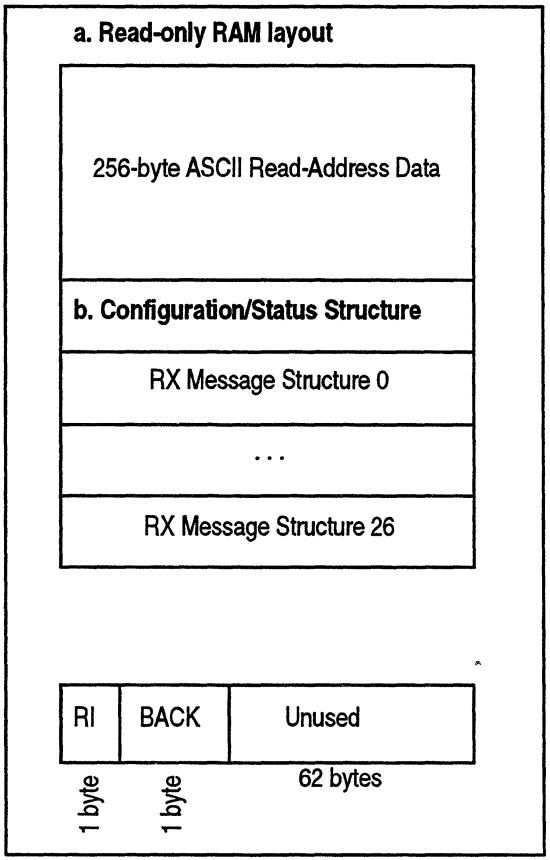


Figure 7-7 Dualport Read-only RAM

Most target systems support byte read accesses to ROM. Burst reading is only a problem if multiple bytes are read from pod 0; other pods do not send interrupts to NetROM. Do not configure NetROM for burst accesses unless you are sure that is what your target is doing. If your target system *is* performing burst accesses, the effective size of the read-address data area is reduced. This is because sending a character to NetROM requires a software handshake (described in "Read-only Target-to-NetROM Message" on page 7-19), and if the target

reads 2 or 4 addresses in quick succession, NetROM will only latch the first address read. Thus, burst reading 2 bytes halves the effective number of interrupt-causing addresses; a burst read of 4 bytes reduces it to a quarter of its original size.

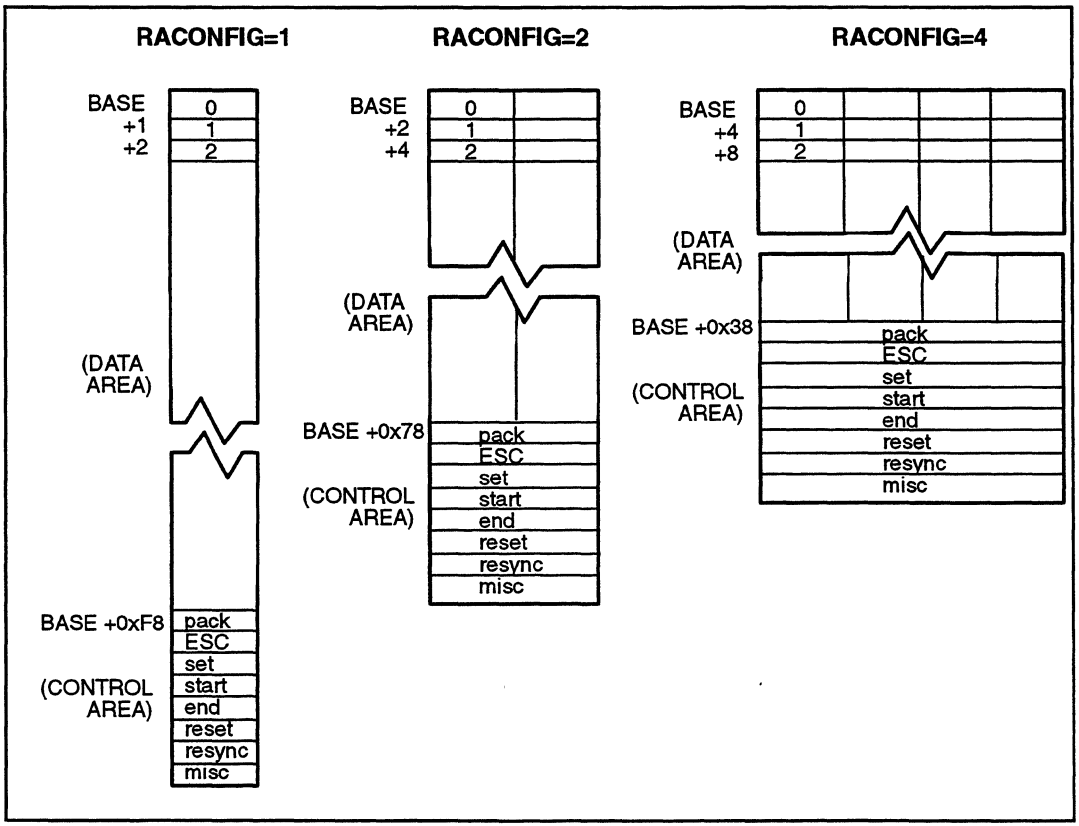


Figure 7-8 Read Address Protocol Interrupt Area

Figure 7-8 shows the layout of the read-address data area for a variety of target types. To configure NetROM for burst reads of pod 0, use the *set raconfig* command.

Part of the data area is set aside to send "out-of-band" information to NetROM. This can include packet delimiter

characters, escape characters, and so on. These characters are described in more detail in "Read-only Target-to-NetROM Message" on page 7-19. The current implementation of the read-address protocol allows for eight out-of-band characters. The first seven have specific functions. The eighth, "misc," will look at the next character to further parse a sub function. The first sub function is "rx_intr_ack." The address of the first out-of-band character is determined by the target's read behavior, as shown in Figure 7-8. The data value corresponding to this address is referred to as the *out-of-band threshold* for the target.

The configuration/status structure of the *readaddr protocol*, shown in Figure 7-7, is used by NetROM to pass data to the target. This structure has three fields; the first is the RI, or Ready-for-Interrupt, byte, which is set to one when NetROM is ready to process, i, readaddrr: protocol interrupts. The target system should *always* make sure this byte is set before sending any data using the *readaddr protocol*, because NetROM will not detect missed interrupts. The second field is the BACK byte, which is incremented by one each time NetROM processes a read-address interrupt, that is, receives a byte from the target. Thus, the target needs to check the RI byte before reading from the data section, and the BACK byte after reading. The third section is reserved and should not be accessed by the target system.

Note since the configuration/status structure is not located in the interrupt-causing read-address data area there is not need to adjust addresses in the structure in response to the target's read behavior. Addresses in the configuration/status structure and the receive message structures are not affected by burst reads from pod 0.

Read-only Target-to-NetROM Message

The flowchart in Figure 7-9 shows how the target can send a message (a single byte at a time) to NetROM. Sending each character has three stages; the first is verifying that NetROM is ready to receive the character. This is done by checking the RI byte in the control/status structure. Note that this must be done twice to prevent possible N-T contention. The second stage is reading the appropriate address in the data section. The offset of the address read from the start of read-address memory provides the value of the byte being sent. (The contents of the data section are undefined and meaningless). The third stage is waiting for NetROM to acknowledge the data just sent by the target. This is done by polling the BACK byte in the control/status section. When the BACK byte is incremented, the data has been received. Note that this must be done twice due to possible N-T contention.

The semantics of the five out-of-band characters described below. Each character is acknowledged in exactly the same way as a normal data character, by an increment of the BACK byte in the configuration/status structure. However, these characters are not passed directly from the target system to the host system; instead, they are interpreted directly by NetROM. Remember that the addresses of these characters is determined by the target system's memory interface hardware. NetROM can be configured for burst reads of pod 0 via the *set raconfig* command.

The remaining three out-of-band characters are reserved and should not be used by the target.

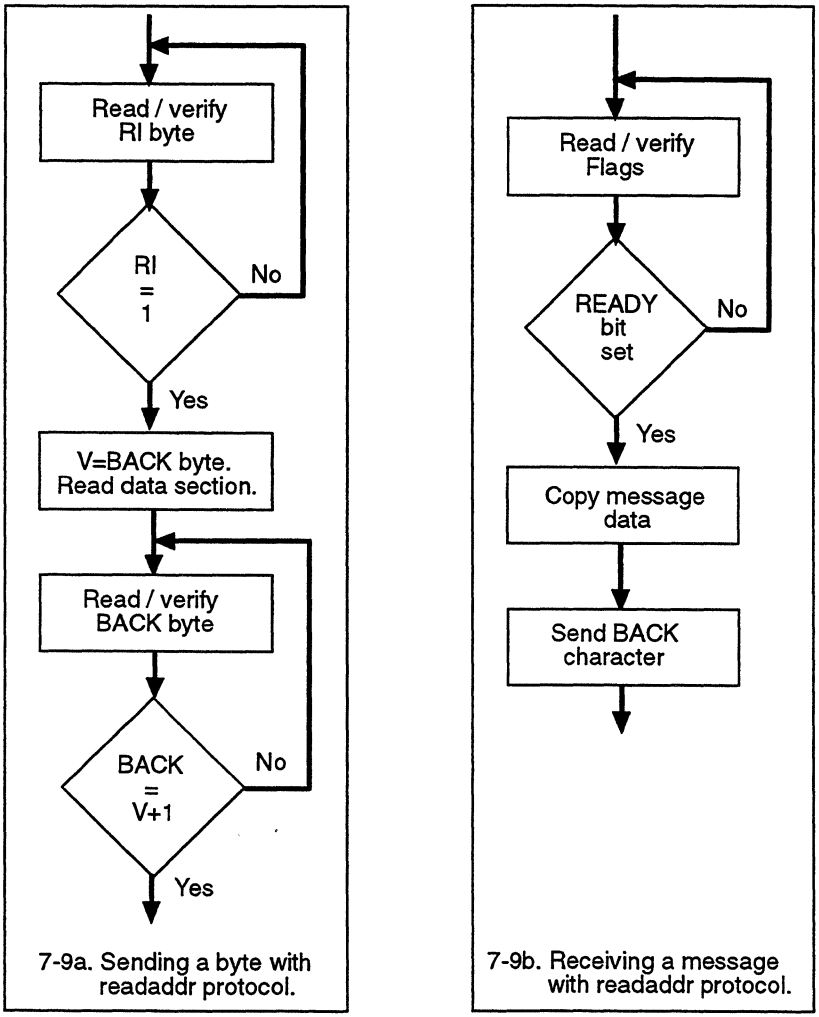


Figure 7-9 Dualport Protocol for Read-only Target Systems

Acknowledging Packets from NetROM

The PACK character is used to acknowledge a block of data sent from NetROM to the target in one of the receive message structures. Sending this character causes NetROM to clear the READY bit in the oldest outstanding message structure. Note that this character is used to acknowledge a single message structure, not a whole message delineated by START and END bits.

Sending 8-bit Data

The ESC character is used when the target wants to send an 8-bit value whose address is greater than the number of available data addresses. For example, most targets can only send values of up to 0xF8, since higher values are interpreted as out-of-band characters. When NetROM receives an ESC character, it will add the value of the out-of-band threshold to the next character received. Thus, in our example, to send a value of 0xFC, the target would send *two* characters: ESC, then 4. On a target which performs 4 burst reads from pod 0, the out-of-band threshold is 0x38. Since 0xFC = (0x38 * 4) + 0x1C, in order to send our example value, this target would have to send the following sequence: ESC, ESC, ESC, ESC, 0x1C.

Setting Emulation Memory

The SET character is used when the target wants NetROM to modify the contents of emulation memory. Modifying emulation memory is often useful for patching code, setting breakpoints, and so on. However, some target systems do not have a write line connected to their ROM sockets and cannot use the external write line for hardware timing reasons. Also, some ROM types, such as the 27c040, cannot be written directly by the target system. Such systems can use the SET character to request that NetROM modify the contents of emulation memory for them.

The SET sequence consists of six characters, as follows: SET, off_0, off_1, off_2, off_3, val. Each character in the sequence is acknowledged as usual, with an increment of the BACK character. Each character following the SET character is an 8-

bit value, and may be sent using an ESC sequence as described above. The first four characters following the SET character represent the offset of the 8-bit value to be written within emulation memory. Offset bytes are sent most-significant-byte first. For example, assume that the target wishes to set memory address 0xbfc163fe to 0xa2. Further, assume that the target does not burst read pod 0 and that its ROM space starts at address 0xBFC00000. Then the offset at which it wishes to set memory is 0x163fe, and the character sequence it should send is: SET, 0x00, 0x01, 0x63, ESC, 0x06, 0xa2.

There is an important caveat when using NetROM to modify emulation memory. If the target is running from ROM it is quite possible that N-T contention during NetROM's write cycle will cause the target to read a bad opcode and crash. To avoid this, the target should jump to RAM prior to sending the value character at the end of the SET sequence. The target should run in RAM until the value character has been acknowledged, then return to running out of ROM. The sample target-side implementation of the dualport driver, described in "An Example Target Implementation" on page 7-23, avoids N-T contention by jumping to RAM. Feel free to obtain the source code if you would like to see how it does this.

Sending Packetized Data to NetROM

The START and END characters are used to delineate a complete message being sent to the host. Normally, NetROM sends characters to the remote host as they are received. On TCP-based connections, this happens when the TCP protocol determines that enough outgoing characters have been queued, or enough time has elapsed, that a packet should be sent. The START and END characters can be used to send an entire block of data to the network interface. On TCP-based connections, this will not necessarily have a dramatic effect, but for host systems using a UDP transport, this allows the target to control the size and frequency of UDP packets sent from NetROM to the host.

Read-only NetROM-to-Target Message

The flowchart in Figure 7-9 depicts the target system receiving a message from NetROM. Note that it is very similar to the target's receive procedure for the readwrite protocol. However, since the target cannot clear the READY bit in an incoming message's Flags field, it must send an "acknowledge" character to NetROM by reading a byte at the address of PACK in the control section.

An Example Target Implementation

The following code is excerpted from a target-side implementation of the dualport mailbox protocols. The full source code is available free of charge from Applied Microsystems with your NetROM unit. This section briefly describes some of the entry points available to the user of the sample code, and then describes in detail the steps needed to port the code to a generic target system.

Table 7-1 Target Implementation Routines

Routine	Description
config_dpram	Initializes control structures and configures the target to use the readaddr; or the readwrite; protocol.
set_dp_blockio	Allows the target programmer to set or clear a bit in the control structure. When set, the interface routines merely poll for data and return if none is present. Otherwise they will wait for data to appear.
dp_chanready	Returns 1 if the NetROM is ready to process messages; 0 otherwise.
ra_putch	Sends a character to the NetROM using the read-address protocol. This routine handles all of the appropriate software handshaking.

Table 7-1 Target Implementation Routines (Continued)

Routine	Description
ra_setmem	Sends breakpoint code to emulated ROM for a read-only target system.
chan_putch	Sends a character to the NetROM using the readwrite protocol. Actually, it stores characters until the message structure is full or chan_flushtx() is called. This reduces protocol overhead.
chan_flushtx	Sends any characters which have been stored but not yet passed to the NetROM. Used only with the readwrite protocol.
chang_getch	Reads a character from the NetROM, if one is present.
ra_getmsg	Reads a complete message from dualport memory, when using the read-address protocol.
chan_getmsg	Reads a complete message from dualport memory, when using the readwrite protocol.
ra_putmsg	Sends a complete message, delineated by the START and END out-of-band characters.
ra_reset	Requests NetROM to reset the target.
ra_resync	Requests NetROM to re-initialize its dual port parameters.
ra_emoffonwrite	Requests NetROM to turn off emulation memory before modifying memory via the ra_setmem call.
chan_putmsg	Sends a complete message, possibly consisting of several dualport message structures, delineated by the START and END bits in the structures flags fields.
ra_rx_intr_ack	Acknowledges receive interrupt.

The sample implementation provides character-oriented input and output, message-oriented input and output, a mechanism to see if data from NetROM is available without reading it (a polling routine), and a mechanism to request that NetROM modify the contents of emulation memory. All routines can be

run in a "blocking" or "non-blocking" mode. The sample implementation supports both the readwrite and the read-address versions of the mailbox protocol. Table 7-1 lists the routines provided in the sample implementation.

Porting the Sample Implementation

The amount of code in the driver which needs to ported to a new target system is very small. Figure 7-10 shows the entirety of the code which requires porting. This code is from the include file *dptarget.h*; in fact, most of the code probably does not need to be changed for a particular target system

```
/* target-native data storage */
typedef unsigned long  uInt32;/* 32 bits unsigned */
typedef unsigned short  uInt16;/* 16 bits unsigned */
typedef short           Int16;/* 16 bits signed */
typedef unsigned char   uChar;/* 8 bits unsigned */
/* prevents private stuff from appearing in the link map */
#defineSTATIC static

/*macro to allow other processes to run in a multitasking sytem
*/
#define YIELD_CPU()

/* Size (in bytes) of the ram-based routine used by read-only
target systems to communicate with NetROM while NetROM sets em-
ulation memory. See routine ra_setmem_sendval().*/
#define RA_SETMEM_RTN_SIZE512

/* dualport access macros */
#defineREAD_POD(cp, addr) \
       (* ((volatile uChar *) ((cp)->dpbase + ((cp)->width *
(addr)) + (cp)->index)))
#defineWRITE_POD(cp, addr, val) \
       (* ((volatile uChar *) ((cp)->dpbase + ((cp)->width *
(addr)) + (cp)->index))) = (val)
```

Figure 7-10 Target-Native Data Storage

The first section of the include file provides target-native storage types. These are used internally to the driver file

dptarget.c and the other include file, *dualport.h*; note that *dualport.h* does not require porting to new platforms.

The STATIC macro is provided to prevent non-entry-point routines from appearing in the target system's link map and conflicting with similarly named routines in the target code. The YIELD_CPU macro is provided for target systems which (a) have a non-preemptive operating system which uses system calls to initiate context switches, and (b) want to use the dualport driver in a blocking mode. Note that blocking mode is disabled by default; to enable blocking, use the `set_dp_blockio()` entry point, described below.

The RA_SETMEM_RTN_SIZE is used for target systems which would like to use the read-address protocol to request that NetROM modify the contents of emulation memory. This macro gives the number of bytes in the `ra_setmem_sendval()` routine, which is copied into a ram buffer before being executed. Consult "Read-only Targets" on page 7-15 to see why this is done. The default value for RA_SETMEM_RTN_SIZE is probably larger than necessary, which will not cause a problem. To "tune" the macro to the size of the routine, you will need to determine the size of `ra_setmem_sendval()` from the link map.

Finally, the READ_POD and WRITE_POD macros do not actually need to be ported, but if a target implementor is willing to sacrifice generality of the sample driver, he or she can improve performance by modifying these macros.

Sample Implementation Entry Points

The sample implementation provides support for both the readwrite protocol and the read-address protocol. To cause code for the appropriate protocol to be compiled, the READWRITE_TARGET macro, the READONLY_TARGET macro, or both, should be defined on the compiler's command line. This section describes the common entry points used for both protocols and the entry points that are specific to each. If the only part of the readaddr protocol being read is that part which sends requests to the NetROM to modify emulation

memory, then both the READONLY_TARGET and the RAWRITES_ONLY macros should be defined.

Common Entry Points

This section describes entry points used by both the read-address and the readwrite protocols. There are a number of references to "channels" in this and subsequent descriptions. These are intended to allow dualport emulation memory to be subdivided into logically separate communications channels, similar to having multiple serial ports. In the current implementation of the driver, only one channel is supported; thus, when a channel number parameter is necessary, use channel 0.

config_dpram

Synopsis

```
int config_dpram(base, width, index, flags, numaccess-
es)
uInt32 base;
int width, index, flags, numaccesses;
```

Description

This routine initializes the dualport driver's internal data structures. It also tells the driver where dualport memory is in ROM space and how to access it. The **base** parameter is the address of the start of dualport memory. The **width** parameter is the number of bytes in a ROM word, and the **index** parameter refers to which byte of the ROM contains pod 0. Bytes are numbered in the "big-endian" order, in which the byte at the word address has index zero. The flags parameter indicates which dualport protocol should be used by the driver; the two choices are DPF_READONLY_TGT and DPF_ONECHANNEL, where the former selects the read-address protocol and the latter selects the readwrite protocol. Finally, **numaccesses** tells the driver how many accesses to pod 0 the target's memory interface hardware will make when reading a single byte.

Figure 7-11 shows the interaction of base, width, index, and **numaccesses** for a variety of target configurations. `Config_dpram()` returns 0 if successful and (-1) if there is some sort of error. This routine should be invoked to configure the dualport interface structures before calling any other driver entry points.

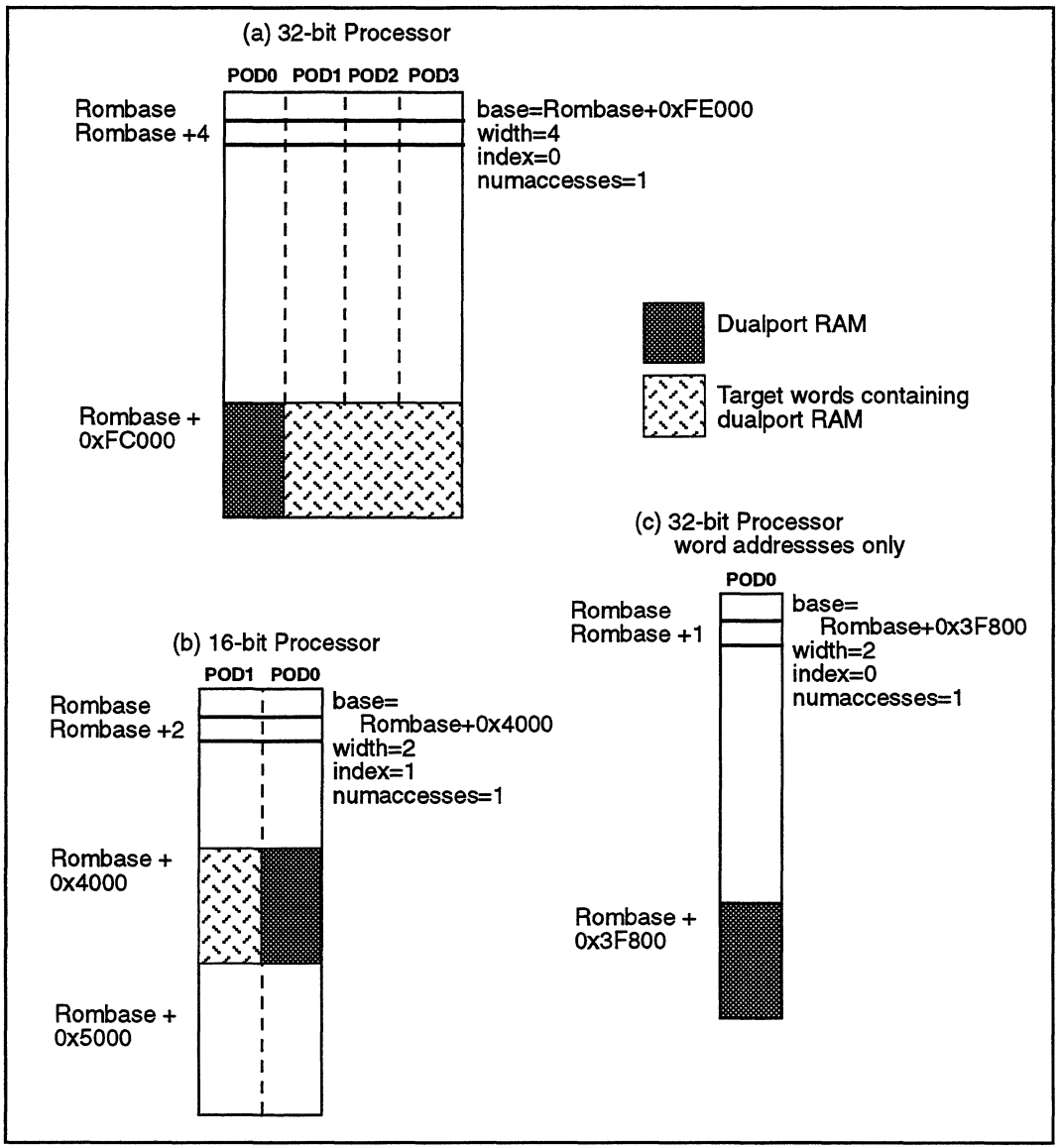


Figure 7-11 Effect of Target Memory Interface in Dualport Protocol

set_dp_blockio

Synopsis

```
void set_dp_blockio(chan, val)
int chan, val;
```

Description

This routine sets or clears the CF_NOWAITIO flag in the driver interface control structure. When set, the driver operates in a "non-blocking" mode. In non-blocking mode, if the transmit or receive routines need to wait for something, they return with an appropriate error code rather than blocking and waiting. When the flag is clear, the driver operates in a "blocking" mode; routines to get and send characters or messages will repeatedly invoke the YIELD_CPU macro rather than return with an error code. If the YIELD_CPU is defined to be a system call, that system call will be performed; if it is defined to be "null" the routine will busy-loop waiting for the event it needs.

The **chan** parameter is the channel affected by the call to `set_dp_blockio()`. The **val** parameter, if nonzero, causes the interface to run in blocking mode; if clear, causes the interface to run in non-blocking mode.

Note

The only channel currently supported is channel 0.

dp_chanready

Synopsis

```
int dp_chanready(chan)
int chan;
```

Description

This routine returns 1 if the dualport channel given by the **chan** parameter is active and if NetROM is ready to use it to communicate with the target. If this routine returns 0, the target should not attempt to perform I/O on the channel. A zero return value may indicate a configuration error on the NetROM side, or it may indicate that NetROM has not received a console path or debug path connection on which to forward data received on the channel.

Note

The only channel currently supported is channel 0.

chan_kbhit

Synopsis

```
int chan_kbhit(chan)
int chan;
```

Description

This routine checks to see if a character is waiting to be read from the dualport protocol interface. If so, it returns 1; if not, it returns 0. This routine will work with either dualport protocol, and will work with character- or message-oriented I/O.

Note

The only channel currently supported is channel 0.

Read-address Protocol Entry Points

This section describes the programmatic interface to the read-address protocol's entry points. Using `ra_setmem()` does not require the target implementor to use the entire read-address protocol. Consult the documentation on the *set rawrites* command for more information.

ra_emoffonwrite

Synopsis

```
int ra_emoffonwrite
```

Description

This routine sends a request to NetROM to turn off emulation memory before modifying memory via a `ra_setmem` call.

Note

Use this routine when you are having trouble with the `ra_setmem` call. For example, if you are unable to set breakpoints in ROM space because the board logic does not release the ROM control to allow NetROM access, call `ra_emoffonwrite` before `ra_setmem()`.
The routine only needs to be called one time. A suggested place to call the routine is after the `config_dpram` function call.

---

ra_getch

Synopsis

```
int ra_getch()
```

Description

This routine reads a character from the read-address receive message structures, if one is available. If one is not, and the interface is in blocking mode, the routine will wait for one to arrive. If in nonblocking mode, the routine returns a -1.

ra_putch

Synopsis

```
void ra_putch(ch)
uChar ch;
```

Description

This routine sends a character to NetROM using the read-address protocol. If NetROM has not set the RI byte in the configuration/status structure for the read-address protocol, `ra_putch()` will wait for it to be set. This routine will handle all ESC sequences which need to be inserted to send the character. When the routine returns, NetROM will have received the character.

ra_getmsg

Synopsis

```
int ra_getmsg(buf, len, bytesread)
uChar *buf;
int len, *bytesread;
```

Description

This routine reads a message from the read-address protocol's receive message structures. The **buf** parameter is a pointer to the receive buffer, the **len** parameter is the number of bytes in the buffer, and **bytesread** is filled in by `ra_getmsg()` with the number of bytes read into the message.

When used in a polling mode, `ra_getmsg()` returns one of four status values: GM_NODATA indicates that no data has arrived since the last poll; GM_MSGCOMPLETE indicates that new data has arrived and that the input buffer now holds the complete message; GM_NOTDONE indicates that data has arrived but that the message is not yet complete; GM_MSGOVERFLOW indicates that more data has arrived, but that it has overflown the input buffer. In a non-polling mode, `ra_getmsg()` will return either GM_MSGCOMPLETE or GM_MSGOVERFLOW.

The following shows an example of using `ra_getmsg()` (or `chan_getmsg()`) in non-blocking mode to receive entire messages.

```
/* reads a message from dualport ram */
int readmsg(buf, lenp)
uChar *buf;
int *lenp;
{   uChar *curbuf;
    int bytesleft, bytesread, status, errcount;
    uInt32 cacr;
    curbuf = buf;
    bytesleft = *lenp;
    errcount = 0;
    status = GM_NODATA;
    while(status != GM_MSGCOMPLETE) {
       bytesread = 0;
#ifdef READONLY_TARGET
       status = ra_getmsg(curbuf, bytesleft, &bytesread);
#else  /* READONLY_TARGET */
       status = chan_getmsg(0, curbuf, bytesleft,
              &bytesread);
#endif /* READONLY_TARGET */
       switch(status) {
       case GM_NODATA:/* nothing present */
              break;
       case GM_MSGCOMPLETE:/* got a complete message */
              bytesleft -= bytesread;
              *lenp = *lenp - bytesleft;
              break;
       case GM_NOTDONE:/* got part of a message */
              bytesleft -= bytesread;
              curbuf += bytesread;
              break;
       case GM_MSGOVERFLOW:/* reset all pointers,
                                          we    ran out of room */
              curbuf = buf;
              bytesleft = *lenp;
              errcount ++;
              break;
       default:
              errcount ++;
              break;
       }
   }
   return(status);
}
```

Figure 7-12 Using ra-getmsg to receive a message

ra_putmsg

Synopsis

```
void ra_putmsg(buf, len)
uChar *buf;
int len;
```

Description

This routine sends an entire message to NetROM using the read-address protocol. The message will be delineated by the START and END out-of-band characters. `Ra_putmsg()` will take care of all ESC sequences that need to be included in the course of transmitting the message.

ra_setmem

Synopsis

```
void ra_setmem(ch, addr, buf)
uChar ch;
uInt32 addr;
uChar *buf;
```

Description

This routine sends a request to NetROM to modify the contents of emulation memory. The **ch** parameter is the 8-bit value to be written. The **addr** parameter is the 32-bit offset from the start of emulation memory to be modified. For example, if ROM starts at 0xFC0000 and the target system wishes to modify address 0xBFC163FE, the **addr** parameter would be 0x163FE. The **buf** parameter is a pointer to a buffer from which the set request will execute. It is necessary to run from RAM during parts of the set request to avoid potential memory contention problems when NetROM executes the write. Consult "Setting Emulation Memory" on page 7-21, above, for more information on the read-address protocol and setting emulation memory.

The `ra_setmem()` routine will send the offset and data value in the order expected by NetROM, and will insert any ESC characters necessary. The buffer pointed to by "buf" must be 32-bit aligned and contain RA_SETMEM_RTN_SIZE bytes of storage. It is the responsibility of the target system implementor to verify that RA_SETMEM_RTN_SIZE is greater than or equal to the size of the driver routine `ra_setmem_sendval()`, which is what is actually copied into the buffer.

ra_reset

Synopsis

```
int ra_reset
```

Description

This routine requests NetROM to reset the target. The routine returns a 0 if the NetROM is not ready or it returns a 1 when done.

Note

The routine may not have time to return when the target is reset.

ra_resync

Synopsis

```
int ra_resync()
```

Description

This routine requests the NetROM to re-initialize (resync) the dual port parameters. The routine returns a 0 if the NetROM is not ready or it returns a 1 when done.

A monitor could use `ra_resync` if it switches from running in EPROM to running in RAM and therefore will re-initialize itself. This routine could be called just before the switch to tell NetROM to re-initialize the dual port parameters so the target and the NetROM will be in sync when the target re-initializes.

ra_rx_intr_ack

Synopsis

```
int ra_rx_intr_ack
```

Description

This routine acknowledges a previous interrupt to the target. The routine returns a 0 if the NetROM is not ready and returns a 1 when done. This routine can help prevent nested interrupts.

This page is intentionally left blank.

Readwrite Protocol Entry Points

This section describes the programmatic interface to the readwrite protocol's entry points. Before using the readwrite protocol, make sure that the NetROM's "groupwrite" environment variable has been set to *readwrite*.

chan_getch

Synopsis

```
int chan_getch(chan)
int chan;
```

Description

This routine reads a character from the readwrite receive message structures for channel **chan**, if one is available. If one is not, and the interface is in blocking mode, the routine will wait for one to arrive. If in nonblocking mode, the routine returns a -1.

Note 

The only channel currently supported is channel 0.

chan_putch

Synopsis

```
int chan_putch(chan, ch)
int chan;
uChar ch;
```

Description

This routine sends a character to NetROM using the readwrite protocol. If a transmit structure is not available on channel **chan** and the routine is running in blocking mode, it will wait for a structure to become available. Otherwise it will return with a status of -1. The routine returns 1 upon success.

Note

The only channel currently supported is channel 0.

chan_getmsg

Synopsis

```
int chan_getmsg(chan, buf, len, bytesread)
int chan;
uChar *buf;
int len, *bytesread;
```

Description

This routine reads a message from the readwrite protocol's receive message structures. The **buf** parameter is a pointer to the receive buffer, the **len** parameter is the number of bytes in the buffer, and **bytesread** is filled in by `chan_getmsg()` with the number of bytes read into the message. The **chan** parameter is the channel on which the message is to be sent.

When used in a polling mode, `chan_getmsg()` returns one of four status values: GM_NODATA indicates that no data has arrived since the last poll; GM_MSGCOMPLETE indicates that new data has arrived and that the input buffer now holds the complete message; GM_NOTDONE indicates that data has arrived but that the message is not yet complete; GM_MSGOVERFLOW indicates that more data has arrived, but that it has overflowed the input buffer. In a non-polling mode, `chan_getmsg()` will return either GM_MSGCOMPLETE or GM_MSGOVERFLOW.

Figure 7-12 shows an example of using `chan_getmsg()` (or `ra_getmsg()`) in non-blocking mode to receive entire messages.

Note

The only channel currently supported is channel 0.

chan_putmsg

Synopsis

```
void chan_putmsg(chan, buf, len)
int chan;
uChar *buf;
int len;
```

Description

This routine sends an entire message to NetROM using the readwrite protocol on channel **chan**. The message will be delineated by the START and END bits in the transmit message structures. The routine will handle breaking large messages into blocks automatically.

Note 

The only channel currently supported is channel 0.

Chapter 8
Virtual Ethernet

Virtual Ethernet is an optional downloadable RAM module for NetROM. Also called *Vether*, Virtual Ethernet gives target systems the ability to become Ethernet communications devices without requiring that they have Ethernet hardware. This feature gives design engineers access to Ethernet communications speed and function during the development cycle even when Ethernet capability will not be needed in the final product. Vether is also useful for debugging target Ethernet hardware and drivers.

Vether operates in a way similar to XLNT Design's Virtual UART in that the target application driver is replaced with a virtual application driver. With Vether, the target's Ethernet driver is replaced with the Virtual Ethernet driver. In this way, communication between NetROM and the target is via Vether, using NetROM's shared memory protocols, and NetROM sends and receives target packets on its Ethernet interface. Figure 8-1 illustrates this process in a logical block diagram.

For additional information about Vether, refer to the NetROM *Application Notes*.

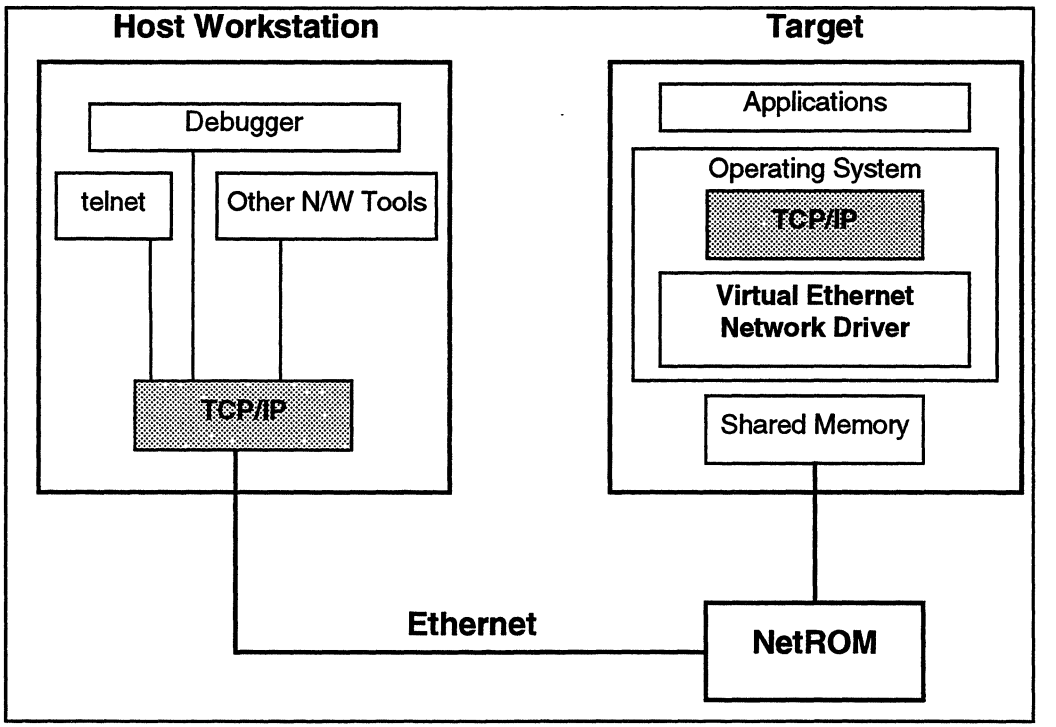


Figure 8-1 Virtual Ethernet Logical Block Diagram

Note

ReadWrite vs. ReadAddr Protocols

NetROM can use two protocols for communication over the ROM address space: ReadAddr and ReadWrite. The ReadWrite protocol requires the target to write to its ROM address space while ReadAddr requires only read access. Because of performance differences between the two protocols, we recommend you use the ReadWrite protocol for target machines that can write to ROM.

Virtual Ethernet Components

Virtual Ethernet functionality is implemented in a target operating system driver and a NetROM RAM module.

Virtual Ethernet Setup Procedure

The setup procedure consists of several steps described below.

- Integrate the Vether driver into the target operating system. This is similar to the process of integrating Applied Microsystems' Virtual UART emulation memory protocol routines. For more information, refer to the *"Virtual Ethernet Application Notes."*
- Download the operating system to NetROM with the *newimage* command or burn it into PROMs.
- Assign the target system a host name and an IP address and add the address to

  `/etc/hosts` or `NIC`
- Select an interrupt line on your target.

  This line must signal a unique interrupt to the CPU. Connect a jumper between one of the NetROM's command pins and the interrupt signal on the target. NetROM will signal the interrupt when it has received a packet for the target.

  The interrupt signal should be active low with a pull up resistor. This is because NetROM does not drive a command pin high, it just disconnects it from ground. If you must use an active high signal, you should connect a 1000 ohm pullup resistor so the signal will be driven high when NetROM disconnects it from ground.

  To enable interrupt signaling, enter the following NetROM command:

  ```
  set tgtctl 1 on rx
  ```

This will cause Net ROM to assert command pin 1 (active low) when a packet arrives.

To enable an active high signal, use

```
set tgtctl 1 off rx
```

NetROM Setup Procedure for Virtual Ethernet

Execute the following commands to setup NetROM. You can add these commands to your batch file so you don't have to enter them each time you bring up NetROM.

- Load the Vether RAM module.

  loadmodule modulepath

- Set the IP address of your target system:

  ```
  setenv tgtip <n.n.n.n>
  ```

- Specify the communication protocol between NetROM and the target:

  ```
  setenv debugpath readwrite
  ```

  -or-

  ```
  setenv debugpath readaddr
  ```

- Allow the target to write to emulation memory if readwrite:

  ```
  setenv groupwrite readwrite
  ```

- Enable Virtual Ethernet and wait for the target to be initialized:

  ```
  setenv vether on
  ```

- Enable interrupt signaling on packet reception:

  ```
  set tgtctl 1 off rx
  ```

  The NetROM side of Virtual Ethernet waits for the target's vether driver to be initialized.

- Reset the target. After the target Vether initializes the target, it synchronizes with NetROM and commences passing packets.

Appendix A
Connector Pinouts

RS-232 Pinouts

Pin	Description
1	Request To Send (RTS)
2	Data Terminal Ready (DTR)
3	Transmit Data (TxD)
4	Ground
5	Ground
6	Receive Data (RxD)
7	Data Set Ready (DSR)
8	Clear To Send (CTS)

Ethernet Pinouts

Pin	IEEE 802.3 Signal	Ethernet II Signal
1	Control In Circuit Shield	Chassis Shield
2	Control In Circuit A	Collision Presence+
3	Data Out Circuit A	Transmit+
4	Data In Circuit Shield	Not Used
5	Data In Circuit A	Receive+
6	Voltage Common	12V Ground
7	Not Used	Not Used
8	Option Shield	Not Used
9	Control In Circuit B	Not Used
10	Data Out Circuit B	Transmit-
11	Data Out Circuit Shield	Not Used
12	Data In Circuit B	Receive-
13	Voltage Plus	+12V
14	Voltage Shield	Not Used
15	Not Used	Not Used

Appendix B
NetROM Processes

Process names and descriptions

This table lists the names of processes commonly encountered in the NetROM environment, a brief summary of the function of each process, and whether it supports multiple instances.

Process Name	Multiple	Description
Console	No	Provides a user interface on the NetROM Console serial port.
conspatbgd	No	Multiplexes data from the target console to host-side listeners.
debugctld	No	Supports direct target control for debug programs
debugpathd	No	Transfers data between the target and the host system.
Kernel	No	NetROM's operating system "process."
netromd	No	Listens for connections on the NetROM Console Port and spawns processes to handle each one.
NetROM Console	Yes	Direct TCP connection providing a non-TELNET command-line user interface.
pingXX (1)	Yes	Sends and receives CMP echo request packets to other network hosts.
snmpd	No	Processes incoming SNMP requests.
telnetd	No	Listens for TELNET connection attempts.
telnetXX(1)	Yes	Provides a TELNET command-line user interface.
TFTP Client(2)	Yes	Downloads a file from a TFTP server.

Notes

(1) XX denotes the number of the process.

(2) The TFTP Client process cannot normally be multiply instantiated.

Appendix C
NetROM Ports and Protocols

Port Addresses

This table lists port addresses on which NetROM listens.

Port Name	Number	Type
BOOTP Client (1)	68	UDP
Debug Control	1237	TCP
Debug Data (2)	1235	TCP
Download (3)	1236	TCP
Upload (4)	1238	TCP
NetROM Console	1234	TCP
SNMP	161	UDP
TELNET	23	TCP
29KJTAG (5)	1239	TCP

Notes

(1)The BOOTP Client Port is only active during NetROM's boot procedure, after NetROM has sent a BOOTP request packet to the network broadcast address.

(2)This port number can be configured using the "debugport" environment variable.

(3)The Download Port must be activated before it can be used.

(4)Activating the Upload Port is required before it can be used.

(5)If the JTAG29K optional RAM module is loaded and enabled.

Appendix D
NetROM Filename Conventions

Batch File Names

NetROM imposes no restrictions on the names of batch files; such files can be named anything convenient for the local operating system. A ".bat" suffix is not necessary, but is often used in examples in this document to improve clarity. Note that TFTP servers running in secure mode require that download files be in a subdirectory of /tftpboot on the server's disk. This directory is implied in all file requests, and should not need to be given explicitly; for example, requesting /tftpboot/startup.bat from a secure server would actually fetch the file /tftpboot/tftpboot/startup.bat from the server's disk.

RARP File Names

If RARP is being used as NetROM's address resolution mechanism, the following conventions must be observed for the NetROM startup file:

- The TFTP server for the startup file must reside at the same IP address as the RARP server.
- The startup file's name must be determined from NetROM's IP address.

The expected filename is the eight-character hexadecimal representation of NetROM's IP address, given in uppercase with no periods and no suffix. For example, if NetROM's address were "192.0.0.210" then the startup file should be named C00000D2. NetROM now makes several attempts to download its startup file. NetROM will then attempt to download the following startup files: "C00000D2," then

"/tftpboot/C00000D2," and finally "tftpboot/C00000D2." After the first successful download, it will proceed with its boot sequence and execute the commands in the startup file. It will not attempt to download other startup files.

Appendix E
NetROM Defaults

Target Console Port

9600 baud

8 data bits

2 stop bits

No parity

No hardware handshaking

XON/XOFF software handshaking disabled

NetROM Console Port

9600 baud

8 data bits

2 stop bits

No parity

No hardware handshaking

XON/XOFF software handshaking disabled

Command Signals

None asserted.

Environment Variables	
batchpath	/tftpboot
consolepath	serial
debugpath	serial
debugport	1235
dprbase	0x3F800
filetype	binary
fillpattern	none
groupaddr	0x00000000
host	(see Note)
loadfile	image.bin
loadpath	/tftpboot
podgroup	0
podorder	0
romcount	1
romtype	27c010
wordsize	8
writemode	flash
verify	on

Note

The "host" variable defaults to the address of the RARP or BOOTP server configured by NetROM's IP address. If NetROM's address is set manually, the default address is "192.0.0.2."

Generic Variables	
consechoon	on
debugecho	on
dplocation	high
emulate	on
raconfig	1
rawrite	off
udpsrcmode	off

Appendix F
Network Basics

This appendix provides a simplified description of network operation. "TCP/IP Network Protocol" on page 1 outlines Transmission Control Protocol/Internet Protocol (TCP/IP), which assembles message packets for network transmission. "Addressing" on page 4 explains network and subnet addressing.

TCP/IP Network Protocol

On Ethernet, Transmission Control Protocol / Internet Protocol (TCP/IP) software allows communication between different networks. A basic system consists of two transceivers connected through a network; see Figure F-1. From the transceiver 1, the application software generates the application data. Next, the TCP software adds the TCP header to the application data forming the TCP packet. Next, the IP software forms the IP packet by adding the IP header to the TCP packet. Finally, Ethernet software creates the Ethernet packet by adding an Ethernet header to the IP packet. The transceiver sends the Ethernet packet across the network to transceiver 2 that receives the packet and successively strips the headers leaving the application data.

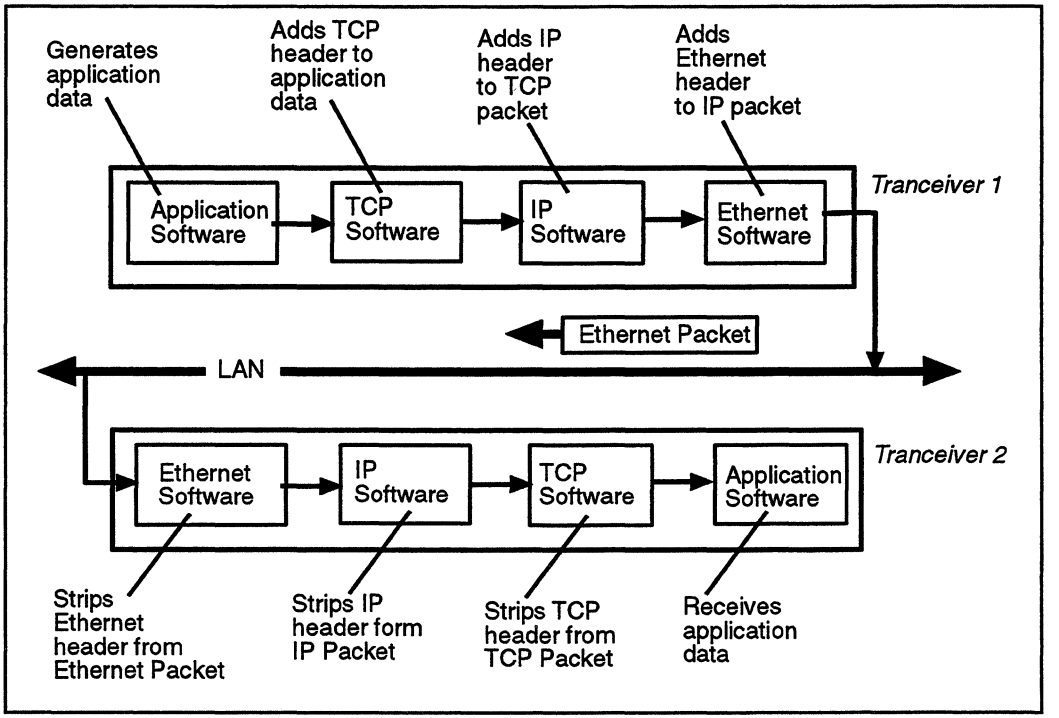


Figure F-1 TCP/IP Network Protocol

Figure F-2 illustrates the Ethernet packet, which consists of a series of embedded data portions with headers. The TCP packet is the data portion of the IP packet, and the IP packet becomes the data portion of the Ethernet packet. Because of this structure, the system can transmit the data within the packet to other networks regardless of the packet type carrying it.

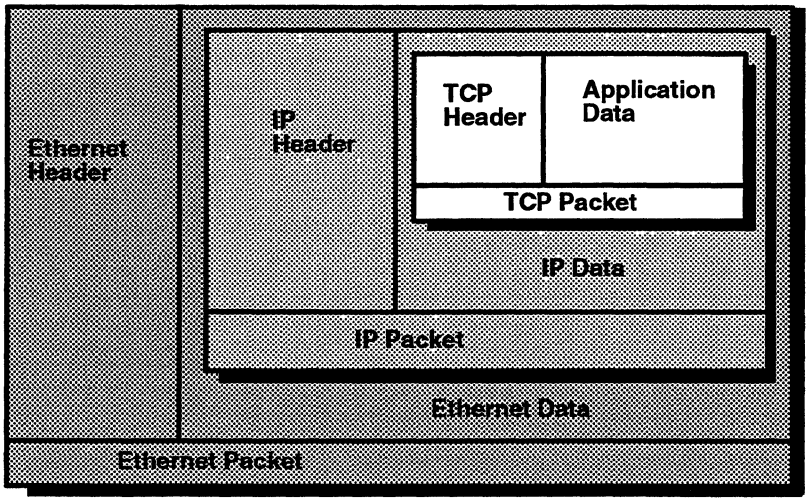


Figure F-2 Ethernet Message Packet

Ethernet Packets

The transceiver routes data across the Ethernet in variable-length "frame" or "packet" format (see Figure F-3). Ethernet packets are self identifying with each packet containing the source address, the destination address, and the information type. Including packet type information in the structure allows the use of multiple protocols on a single machine or on the same physical network without interference.

Preamble	Destination Address	Source Address	Packet Type	Data	CRC
64 bits	48 bits	48 bits	16 bits	368-12,000 bits	32 bits

Figure F-3 Ethernet Packet Format

Besides the destination and source addresses, Ethernet packets contain this additional information.

Preamble	Consists of alternating 0s and 1s to aid the receiving transceiver in signal synchronization. The last two digits are 11.
Packet Type	Informs the receiving transceiver protocol software module used to process the packet. In our case this is Internet Protocol.
CRC	Used to identify data errors within the packet.

Addressing

A host is any end-user computer connected to the network. Each host under TCP/IP has a physical (Ethernet) address and an Internet (IP) address.

Physical / Ethernet Addresses

Each host's Ethernet interface contains the host unit's physical or Ethernet address stored in a ROM or a PAL. The physical address allows the computer to determine the packets the computer will receive. The physical address is a six-byte, hexadecimal number. Ethernet hardware manufacturers purchase blocks of physical address and assign the address in sequence as they manufacture the interface hardware. Physical addresses can not be changed.

Note

The physical address relates to the interface hardware rather than the host. Replacing the host's interface hardware changes the host's physical address.

Internet Addresses

An Internet (IP) address is a 4-Byte number written in dotted decimal notation (XX.X.XX.XXX). This notation expresses each byte as a decimal integer between 0 and 255 with decimal points separating the bytes.

For example: 10.4.25.196

The Internet address differs from the physical address because users can assign and program the Internet address.

Caution

You must insure the Internet address is not duplicated anywhere on the network, otherwise duplicate Internet addresses can seriously interfere with network operation.

The Internet address consists of two portions: a Network ID portion and a Host ID portion. The Network ID identifies a given physical network, and the Host ID identifies a particular host on that network.

There are five defined Internet address classes. Each class is divided based on the number of networks and the number of hosts. Only three of the classes (A, B, and C) can specify individual hosts (Figure F-4).

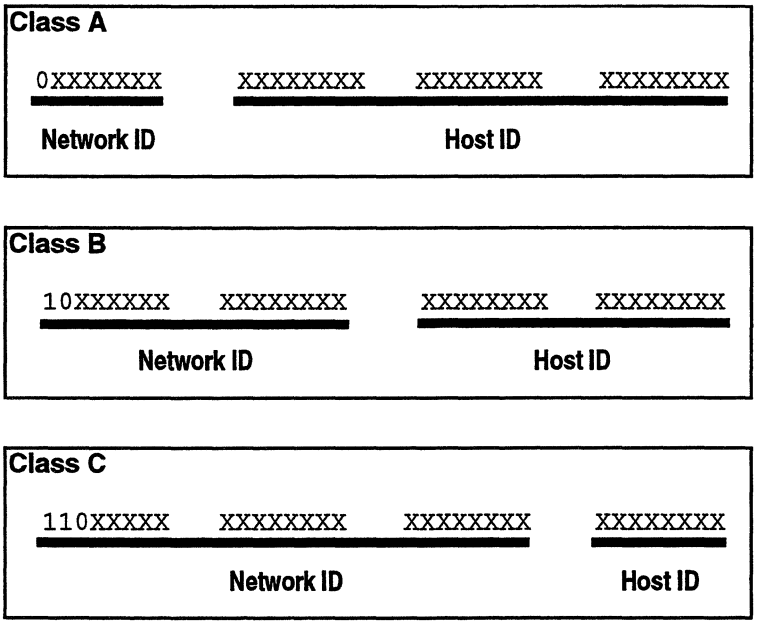


Figure F-4 Internet Address Classes

Class A

Networks having more than 65,536 (2^{16}) hosts use class A addresses. The first eight bits are the network ID, and 24 bits are the host ID. The high-order bit of the first byte (network ID) must always be 0, allowing network addresses from 0 to 127.

Note

Network ID 127 should not be used because it is reserved for the local and loopback device.

Class B

Networks have between 256 (2^8) and 65,536 (2^{16}) hosts use class B addresses. The first 16 bits of the address are the network ID, and the second 16 bits are the host ID. The high-order bits of the first byte (network ID) must be 10 allowing network addresses between 128.0 and 191.255.

Class C

Networks with fewer than 256 (2^8) hosts use class C addressing. The first 24 bits of the address are the network ID, and the remaining 8 bits are the host ID. The high-order bits of the first byte (network ID) must be 110 allowing network addresses from 192.0.0 to 223.255.255.

Network Addresses

Convention regards the address of the network itself to have a host ID of 0. Therefore, the user should never assign others host a host ID of 0. A class B network address would have the form:

x.y.0.0

For example: 192.12.0.0.

Broadcast Address

The Ethernet standard provides for a Broadcast Address that refers to all hosts on the network. Broadcast addresses allows a copy of a message packet to be sent to all hosts on the system. An Internet address having a host ID of all bits set to 1 (255) is the broadcast address. A class B broadcast address would have the form:

x.y.255.255

For example: 192.12.255.255.

Subnets

As the number of physical networks in a given class grows, the pool could run out of addresses. To reduce the number of addresses needed in a large system Subnet Addressing and Subnet Routing was developed. This allows a group of physical networks (subnets) to share a single network address.

To do this the Internet address is replaced with an Internet component and a local component. The Internet component is the network address shared by the various local networks.

The local component consists of two parts. One part identifies a particular subnet, and the second part identifies a host on that subnet. The subnet address is concealed in the host ID portion of the Internet address (Figure F-5).

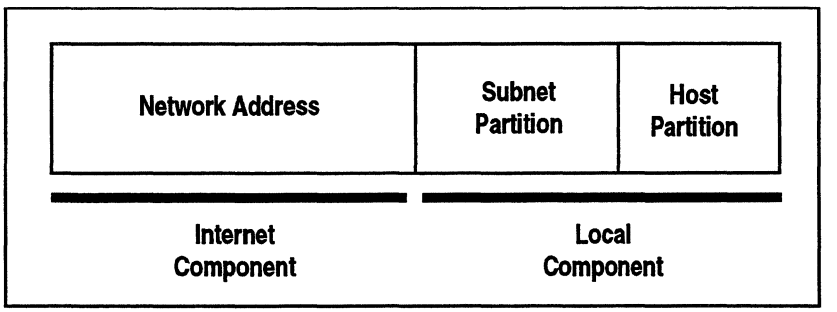


Figure F-5 Internet Address With Subnet Addressing

Figure F-6 illustrates how subnet addressing works with a class B network. In this system, the third byte of each Internet address specifies the subnet (for example: 192.16.1.2), and the fourth byte identifies the host (for example: 192.16.1.2). To the rest of the Internet, the subnets and their hosts appear to be a collection of hosts on network 192.16.0.0. The local gateway accepts all packets with this network address and routes them to the appropriate subnet by examining the third byte of the address. The fourth byte of the address identifies the host on the given subnet.

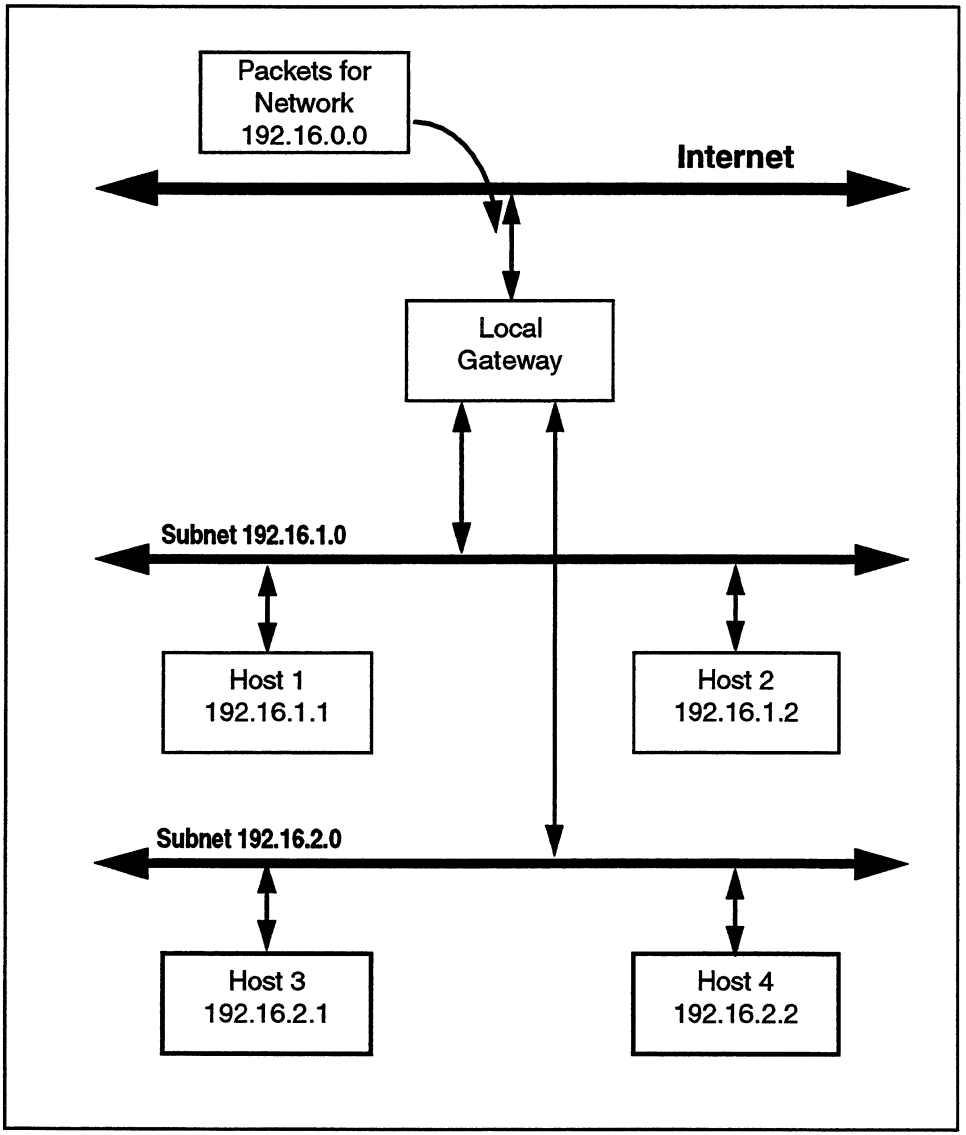


Figure F-6 Subnet Addressing

The system shown in Figure F-6 uses the first 8 bits of the 16 bit local component to identify the subnet (for example: 192.16.1.2), and the remaining 8 bits identify the host (for example: 192.16.1.2). This system allows up to 256 subnets with 256 hosts per subnet.

This is not the only method of local component partitioning. Each network administrator can partition the local component in any manner that is usually based on expected growth of the system. For example, if a network administrator determines that a class B site requires many subnets with a few hosts on each subnet, they can allocate 12 bits to the subnet partition and 4 bits for the host partition. This example allows 4096 (2^{12}) subnets with up to 16 (2^4) hosts per subnet (including network and broadcast addresses).

Subnet Masks

The Internet standard requires every site using subnet addressing to define a 32-bit subnet mask for each physical network at the site. A network's subnet mask defines how. Bits in the subset mask are set to 1s if the corresponding bits (including the Internet component) in the Internet address are considered to be part of the network address. The bits are set 0s if the corresponding bits in the Internet addresses are treated as part of the host identifier.

If a network administrator of a class B network wants to allocate 12 bits of the local component to the subnet partition and 4 bits to the host partition, the required subnet mask would be shown in Figure F-7.

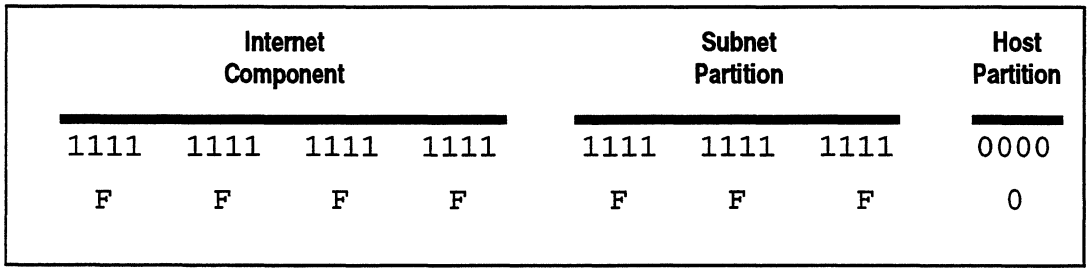


Figure F-7 Subnet Mask for a Sample Class B Network

Since the Internet component and the subnet partition together identify a particular physical network, they comprise the "network address," and all bits of these two fields are set to 1s in the subnet mask. The host partition identifies a particular host on a given network; all bits of this field are set to 0s in the mask. The subnet mask for Figure F-7 would be "255.255.255.240."

Each host on a network has a copy of the subnet mask for that network. By using the subnet mask, the host can extract the network address from an Internet address; this helps make packet-routing algorithms efficient.

Routers

Routers connect two or more networks and transfer message packets among those networks. The router uses software algorithms and routing tables to make the routing decisions necessary to ensure each packet reaches its destination. Often, the terms router and gateway will be used interchangeably.

Router Tables

Both hosts and routers use routing tables to determine where to send message packets. Router tables do not contain complete information on how to reach each destination address. Instead, the tables tell the host or router where to send the packet to reach the next step along the path to the packet's destination. Essentially, the routing table contains pairs of addresses (D

and R). "D" represents the destination's network address, and "R" represents the Internet address. These addresses may point to a router or directly to a host connected to the same subnet.

When a host or router needs to forward a message packet, it extracts the destination's network address by performing a logical AND operation between the network address and the local subnet mask. Next, the host or router searches its routing table for the matching destination network address (D). If a match is found, the host or router sends the message packet to the corresponding destination (R). In turn, the host or router at the destination performs the same process and routes the message packet to the next host or router. If the destination lies on the same subnet as the router or host, the router delivers the packet directly to the destination.

For example, host 4 needs to send a message packet to host 7 (see Figure F-8). Since all the networks are class B, the subnet mask for each network is 255.255.0.0 (binary: 11111111 11111111 00000000 00000000). Also, host 4 contains the routing table shown in Table F-1.

Table F-1 Example Host 4 Routing Table

Destination Network (D)	Routing Address (R)
192.7.0.0	192.12.0.4
192.12.0.0	Deliver Directly
192.16.0.0	192.12.0.4
192.41.0.0	192.12.0.4

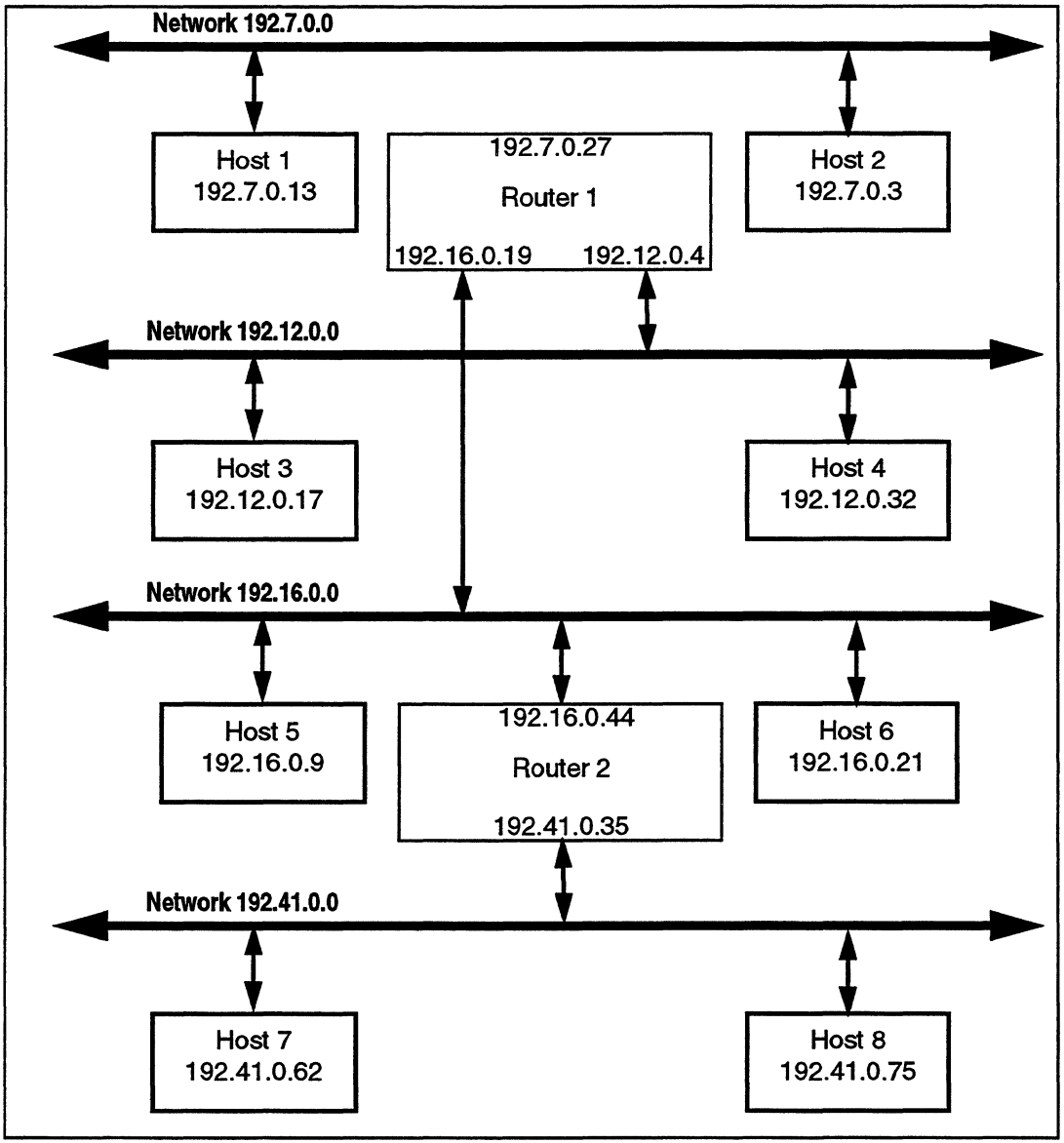


Figure F-8 Example Network with Routers

Host 4 begins constructing the message packet and addressing it to host 7 (Internet address: 192.41.0.62). Host 4 extracts the network address (192.41.0.0) by ANDing the host 7 Internet address with the subnet mask.

Subnet Mask	FF FF 00 00
Internet Address =	CO 29 00 3E
Destination Address	CO 29 00 00

This address matches a destination network address in the routing table (Table F-1) with a corresponding routing address of 192.12.0.4. Host 4 sends the message packet to 192.12.0.4 (router 1).

Router 1 receives the packet and extracts the destination network address (192.41.0.0). Router 1 searches its router table (Table F-2) for the destination address and its corresponding routing address which tells router 1 to send the packet to 192.16.0.44 (router 2).

Table F-2 Router 1 Router Table

Destination Address	Routing Address
192.7.0.0	Deliver Directly
192.12.0.0	Deliver Directly
192.16.0.0	Deliver Directly
192.41.0.0	192.16.0.44

Router 2 receives the packet and extracts the destination network address (192.41.0.0). Router 2 searches its router table (Table F-3) for the destination address that tells router 2 to send the packet directly to host 7 at address 192.41.0.62.

Table F-3 Router 2 Router Table

Destination Address	Routing Address
192.7.0.0	192.16.0.19
192.12.0.0	192.16.0.19
192.16.0.0	Deliver Directly
192.41.0.0	Deliver Directly

In summary, the router receives a packet, extracts the destination network address, finds its destination address in its router table. The routing table tells the router to send the packet directly to a host or to another router. In the example, router 1 was unable to send the packet directly to the host, therefore, router 1 sent the packet to router 2. Router 2 could send the packet directly to the destination host 7 (see Figure F-9).

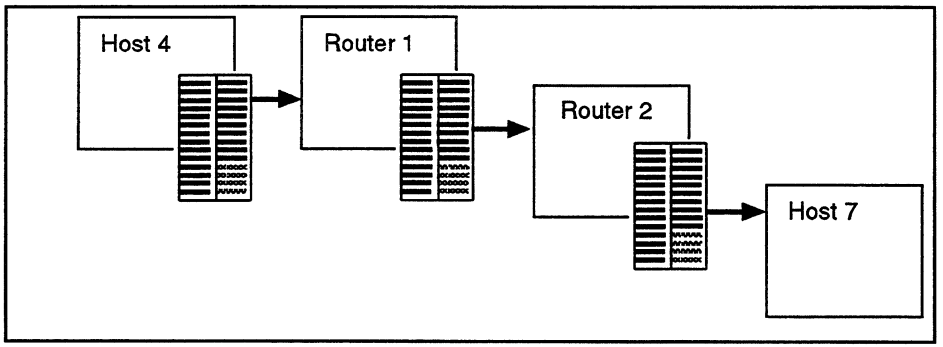


Figure F-9 Example Message Packet Flow

Glossary

Address Resolution	The process of establishing a mapping between an Ethernet address and an IP address.
ARP	Address Resolution Protocol, used to determine a destination host's Ethernet Address using its known IP address.
BOOTP	An address resolution protocol which can also supply the name of a startup file.
Client	A requestor of a service from some provider on the network; e.g. a BOOTP client, which sends out BOOTP requests to determine its own IP address.
Connection	An IP connection is determined by a 4-tuple definition: <source IP address, source port, destination IP address, destination port>. This is to allow more than one dialog between two hosts, using either the same source or the same destination port number.
Console Path	The route by which the host system establishes a console session with the target.
Debugger	A program which runs partly on the target system and partly on the host, for the purpose of providing a user-friendly interface to engineers debugging the target system.
Debug Path	The route over which debugger packets travel between the host system and the target.
DIP	Dual In-Line Package.
Download Path	The route over which emulation images are sent from the host system to the target.
Dumb Terminal	A monitor and keyboard system lacking a significant CPU; "dumb" terminals use RS-232 serial ports to provide an interface to systems which do not come equipped with a monitor.

EPROM	A form of ROM which can be erased and reprogrammed.
Gateway	In IP routing, a gateway is a host on the local network which agrees to route packets to other networks.
Host	A computer on an IP network.
Host, System	A computer which the embedded systems engineer uses to develop code, run debuggers, or establish NetROM sessions.
Internet	A large network of heterogeneous sub-networks.
IP	Internet Protocol, the protocol used to send packets between nodes on the Internet.
IP Address	A 32-bit value, often represented with each byte in decimal and separated by a period.
LED	Light-Emitting Diode.
Path	A route for information to travel from the host system to the target, or from the target to the host system; a file address, consisting of a tree structure for reaching a particular file in a computer memory.
PLCC	Plastic Leaded Chip Carrier.
Plug	The end of an emulation pod which is inserted into a ROM socket on the target system.
Pod	Emulation memory on NetROM, the physical cable leading from NetROM to the ROM socket on the target, or both.
Pod Group	One or more ROMs used together to emulate "words" of ROM memory on the target system.
Port	A TCP or UDP identifier, to distinguish between different destinations using the same protocol.
RARP	Reverse Address Resolution Protocol, an address resolution protocol used by NetROM to determine its own IP address using its known Ethernet address.
ROM	Read-only memory.

Route	In the IP sense, a route is uniquely identified by a destination host, a gateway to that host, and a metric describing "how hard" it is to get to the destination through the gateway.
RS-232	A common serial line protocol.
Server	A provider of some service on the network; e.g. a TFTP server which responds to TFTP requests.
Session	The data exchanged over some kind of communication connection; for example, a TELNET login to NetROM constitutes a terminal session.
Socket	When referring to ROMs, the receptacle into which the ROM is inserted on the target system. When referring to network communications, sockets are an operating system interface which provides a communications end point for TCP or UDP.
SLIP	Serial Line IP, a version of IP which runs over serial links.
SNMP	Simple Network Management Protocol.
Subnet	A "discrete" network, such as an Ethernet LAN, which is part of the larger Internet.
Subnet Mask	A 32-bit value whose logical-and with an IP address determines whether a particular address is on a particular subnet.
Terminal Session	See *session.*
Target System	The computer whose ROMs are being emulated by NetROM.
TCP	Transmission Control Protocol, a connection-oriented end-to-end transport protocol running on top of IP.
TCP	See *connection.*
TELNET	A terminal emulation protocol.
TFTP	Trivial File Transfer Protocol, used by NetROM to request files to download.

UDP	User Datagram Protocol, a connectionless end-to-end transport protocol running on top of IP.
UDP	See *connection*.
XON/XOFF	A software handshaking protocol used on RS-232 lines.

Index

D

E

F

G

H

I

L

M

N

P

R

S

T

U

W