

# NetROM™ 500 Series

## User's Manual

P/N 924-07001-01



Applied Microsystems Corporation



Applied  
Microsystems  
Corporation

# **NetROM™ 500 Series**

## **User's Manual**

P/N 924-07001-01

**May 1997**

Copyright © 1996 Applied Microsystems Corporation.  
All rights reserved.

Information in this document is subject to change without notice. Applied Microsystems Corporation reserves the right to make changes to improve the performance and usability of the products described herein.

Applied Microsystems Corporation's CodeTAP and SuperTAP products are protected under U.S. Patents 5,228,039 and 5,581,695. Additional patents pending.

### **Trademarks**

CodeTAP is a registered trademark of Applied Microsystems Corporation.

The following are trademarks of Applied Microsystems Corporation: CodeCONNECT, CodeICE, CodeTEST, CPU Browser, NetROM, NSE, RTOS-Link, SuperTAP, Transparent Breakpoints, VSP-TAP

Other product names, trademarks, or brand names mentioned in this document belong to their respective companies.

# Contents

---

## Chapter 1 Introduction

NetROM features .....	1-2
Memory emulation .....	1-2
Target communication .....	1-3
Target control .....	1-3
User interface .....	1-3
Debugger integration .....	1-3
Embedded systems development environment.....	1-5
Standard development environment .....	1-6
Development environments using NetROM.....	1-6
Documentation overview .....	1-8
Documentation conventions .....	1-10
Warnings, cautions, notes .....	1-10
Radio interference warning .....	1-11
Operating requirements .....	1-12
Standard electrostatic precautions .....	1-12
Support services .....	1-13

## Chapter 2 NetROM Services

NetROM console .....	2-2
Communications paths.....	2-3
Debug path.....	2-4

Console path.....	2-7
Channel paths.....	2-9
Download path.....	2-11
Emulation memory.....	2-11
ROM groups.....	2-12
Configuring ROM groups.....	2-15
Downloading ROM groups.....	2-15
Optional downloadable RAM module .....	2-16
Command and status signals.....	2-16
NetROM LEDs.....	2-18
Main unit .....	2-18
Active cables .....	2-19

### **Chapter 3**

#### **Hardware Installation**

Collecting equipment.....	3-2
Connecting power .....	3-5
Connecting AC power cord.....	3-5
Connecting to Ethernet .....	3-6
Connecting ROM emulation cables.....	3-8
Cable configurations .....	3-8
Connection steps .....	3-12
Connecting NetROM console serial port .....	3-27
Connecting target serial port.....	3-27
Connecting the write signal .....	3-29
Connecting the reset signal.....	3-31
Connecting the command status connector.....	3-33

**Chapter 4**  
**Software Installation**

Installing the software .....	4-2
PC .....	4-2
Sun, HP workstations .....	4-2
Updating the software .....	4-3
Establishing network communications .....	4-3
Specifying NetROM startup file .....	4-6

**Chapter 5**  
**User Interface**

NetROM command line processing .....	5-2
Processes .....	5-2
Terminal control characters .....	5-4
History substitution .....	5-5
Batch processing .....	5-6
Environment variables .....	5-7
NetROM commands .....	5-9
Understanding the command descriptions.....	5-11
Network interface commands .....	5-13
Target interface commands .....	5-21
Process control commands .....	5-32
Set commands .....	5-35
Display commands.....	5-55
ROM set commands.....	5-78
Miscellaneous commands.....	5-87
Environment variable commands.....	5-101

**Chapter 6**  
**Utilities**

Understanding the utility descriptions..... 6-2

**Chapter 7**  
**Debugger Support**

NetROM debug paths ..... 7-1

Passing data across the debug path ..... 7-2

The debug control port ..... 7-3

Debug control functions..... 7-3

**Chapter 8**  
**Alternate NetROM Interfaces**

Non-TELNET terminal sessions..... 8-2

**Chapter 9**  
**Emulation Memory Mailbox Protocol**

Communication driver API ..... 9-2

Porting the driver ..... 9-3

Entry points..... 9-7

**Chapter 10**  
**General Porting Guide**

Introduction ..... 10-1

Porting overview ..... 10-2

Process..... 10-3

Overview ..... 10-7

Procedure ..... 10-8

NetROM batch files ..... 10-23

Batch file for MVME 162.....	10-23
Batch file for Cogent PowerPC 603 (CMA277).....	10-25
Batch file for Motorola/Cogent 68040 IDP target .....	10-27

**Chapter 11**  
**Virtual Ethernet**

Virtual Ethernet components .....	11-2
Virtual Ethernet setup procedure.....	11-3
NetROM setup procedure for virtual Ethernet .....	11-4

**Appendix A**  
**Connector Pinouts**

RS-232 pinouts .....	A-1
Ethernet pinouts.....	A-2

**Appendix B**  
**NetROM Processes**

Process names and descriptions .....	B-1
--------------------------------------	-----

**Appendix C**  
**NetROM Ports and Protocols**

Port addresses .....	C-1
----------------------	-----

**Appendix D**  
**NetROM Filename Conventions**

Batch file names .....	D-1
RARP file names.....	D-1



**Appendix E**  
**NetROM Defaults**

**Appendix F**  
**Mailbox Protocol Implementation**

Sharing emulation memory.....	F-1
Memory contention issues .....	F-2
Dualport emulation memory .....	F-4
The dualport message structure .....	F-6
Read-address memory.....	F-9
Dualport protocol.....	F-10
Target-to-NetROM message .....	F-12
NetROM-to-target message .....	F-12

**Glossary**

# Chapter 1

## Introduction

---

NetROM is a universal debugging platform providing high-speed target communication and debugging functionality for use in embedded system development. NetROM acts as a link between your preferred debugger and target monitor to provide faster downloads, remotely control the target, and emulate ROM memory devices. NetROM requires almost no target resources and can be rapidly moved from project to project and from processor to processor.

NetROM's only required physical connection to your target system is via a ROM socket or a ROM bus header. Using the high-speed data-transfer rates available on Ethernet LANs, NetROM updates emulation memory with new images much more quickly than with conventional serial- or parallel-link ROM emulators.

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 can be data-packet transfers between the target system and a remote host program.

NetROM gives developers convenient communications paths to target systems via a serial link and a virtual UART 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 functionality has been integrated into many existing debuggers and target monitors or agents. Through the use of virtual UART technology, NetROM provides the debugger target access through Ethernet and the ability to control

program execution from the debugger user interface. Now you can set breakpoints in the ROM address space of your target.

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

### Memory emulation

- 1 MB or 4 MB of emulation memory. Can divide memory into 4 pods, each emulating up to 1 MB, 2 pods, each emulating up to 2 MB, or 1 pod, emulating up to 4 MB.
- Emulation of 64K, 256K, 512K, 1 MB, 2 MB and 4 MB ROMs.
- Support for 8-, 16-, and 32-bit words.
- Support for 64-bit words by using more than one NetROM unit.
- Support for more than 4 MB by using more than one NetROM unit.
- Simultaneous emulation of multiple ROM types and word sizes by using different pods.
- Support for 28-, 32-, and 40-pin DIPs, 32- and 44-pin PLCC sockets, 50- and 60-pin headers, and TSOP and PSOP surface mount packages.
- Support for both 5V and 3.3V memory devices automatically.

## Target communication

- ❑ High-speed Ethernet connection between target and host using no target resources.
- ❑ Rapid code downloads at up to 50K/sec over Ethernet, using standard protocols, such as TFTP or TCP.
- ❑ Four virtual UART channels providing four LAN channels for multiple user sessions with NetROM or the target.
- ❑ Communication with target systems using RS-232 and emulation memory mailbox.
- ❑ Address resolution using BOOTP or RARP.
- ❑ Network manageable by SNMP.

## Target control

- ❑ Eight status signals from the target that can be polled at will.
- ❑ Eight target command signals that the user can assert.

## User interface

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

## Debugger integration

- ❑ Integrated with multiple source-level debuggers. Application notes on the Applied World Wide Web page (<http://www.amc.com>) describe the steps required to utilize NetROM features with your debugger.
- ❑ 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 module to support optional Virtual Ethernet feature.

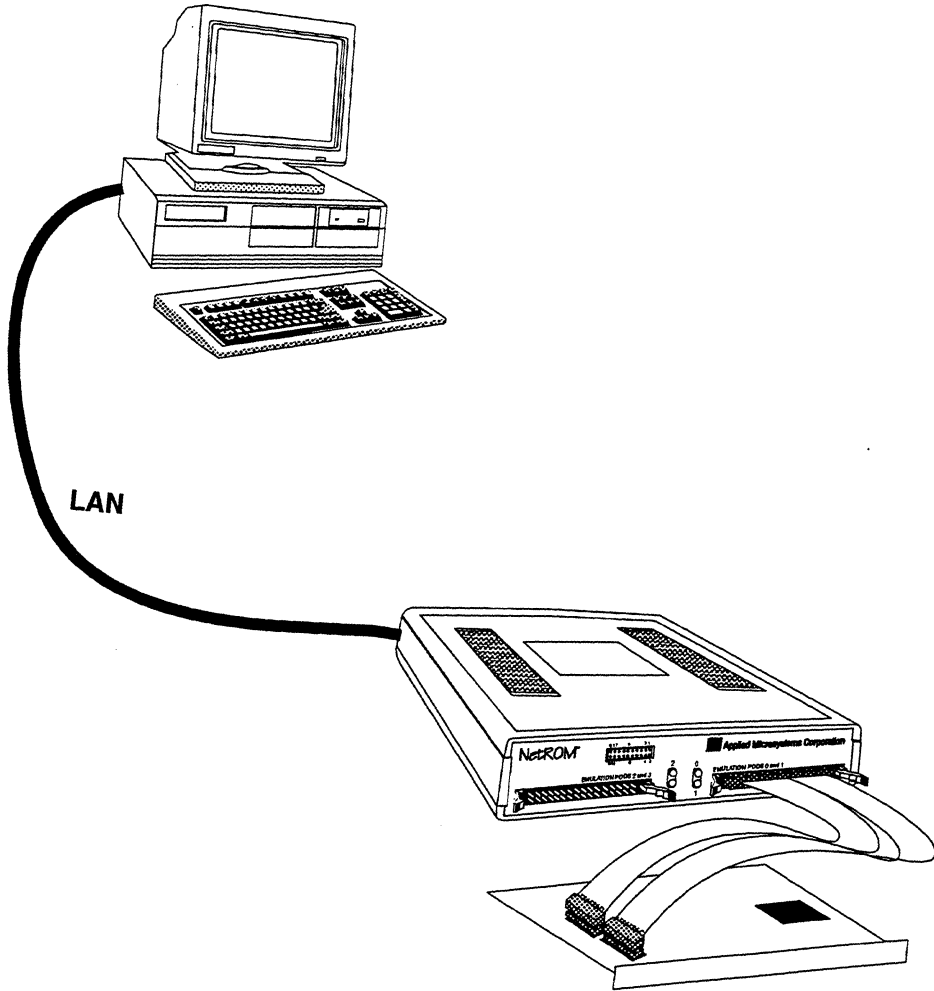


Figure 1-1 NetROM system

---

## 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 target control device such as an in-circuit emulator (ICE), NetROM, or JTAG.

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. NetROM improves your productivity in the download and debug phase by approximately 20%. Although NetROM's ROM emulation features are powerful, NetROM's most important function is communication between the development host and the target system. 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

<b>Path</b>	<b>Description</b>
Download	Mechanism in which an image file created on the host system becomes accessible on the target system.
Console and Channel	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 ROM emulators, which obviously can take considerable time over a serial line.

## Development environments using NetROM

NetROM technology addresses the drawbacks of a traditional RS-232 development environment. In addition to emulating ROMs, EPROMs, and flash memory devices, NetROM functions as a communications nexus between the host computer and the target system. Because NetROM connects to a high-speed



Ethernet LAN, it can multiplex all of the different communications paths from their respective sources onto the network.

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

NetROM is powerful. 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.

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 Hardware Installation**

Gives step-by-step instructions for installing the NetROM hardware.

### **Chapter 4 Software Installation**

Gives step-by-step instructions for installing the software and setting up communications.

### **Chapter 5 User Interface**

Describes the NetROM user interface, including the command set and environment variables.

### **Chapter 6 Utilities**

Describes NetROM utilities.

### **Chapter 7 Debugger Support**

Describes NetROM's debugger support features.

### **Chapter 8 Alternate NetROM Interfaces**

Describes how users can write their own programs to interface to NetROM.

### **Chapter 9 Emulation Memory Mailbox Protocols**

Describes the virtual UART emulation memory mailbox protocol.

**Chapter 10 General Porting Guide**

Describes procedures for porting your target monitor or operating system to NetROM.

**Chapter 11 Virtual Ethernet**

Introduces Virtual Ethernet, an optional downloadable RAM module to NetROM, and provides installation instructions.

**Appendix A Connector Pinouts**

Lists the RS-232 and Ethernet connector signals.

**Appendix B NetROM Processes**

Describes the common NetROM processes.

**Appendix C NetROM Ports and Protocols**

Lists the NetROM port addresses.

**Appendix D NetROM Filename Conventions**

Describes the filename conventions for batch and RARP files.

**Appendix E NetROM Defaults**

Lists port and environment defaults.

**Appendix F Mailbox Protocol Implementation**

Addresses issues associated with implementation of emulation memory mailbox protocol.

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.

In addition, the Applied web site (<http://www.amc.com>) provides application notes and updated technical information.

---

## 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.

---



---

## Radio interference warning

This equipment generates, uses, and can radiate radio frequency energy.

Caution



---

This instrument is intended for use in the development of microprocessor-based systems. At this stage of development, these target devices typically include no inherent design to limit the emissions of electromagnetic energy.

Precautions should be taken to prevent harmful radiation to radio communications and other nearby sensitive electronic systems by means of isolation, separation, or shielding, where necessary.

Use of this instrument in a residential area is likely to cause harmful interference, in which case, the user will be required to correct the interference at his own expense.

---

### FCC Rules

It is temporarily permitted by regulation and has not been tested for compliance with the limits of Class A computing devices pursuant to Subpart J of Part 15 of FCC Rules, which are designed to provide reasonable protection against such interference.

### EMC Directive

For compliance to the essential requirements of the EMC Directive 89/336/EEC, if a ground post is provided on the back of the chassis or on the external power supply, a properly bonded ground strap must be connected to it.

---

## Operating requirements

Before setting up NetROM, you should determine where you want to install NetROM and to make sure that the operating environment is prepared.

## Standard electrostatic precautions

This instrument contains static-sensitive components that are subject to damage from electrostatic discharge. Use standard ESD precautions when transporting, handling, or using the instrument or when connecting/disconnecting the instrument and the target,

Applied Microsystems recommends the use of the following precautions:

- Use wrist straps or heel bands with a 1 Megohm resistor connected to ground.
- On the work surface and floor, use static conductive mats with a 1 Megohm resistor connected to ground.
- Keep high static-producing items, such as non-ESD-approved plastics, tape and packaging foam, away from the instrument and the target.

The above precautions should be considered as minimum requirements for a static-controlled environment.

**Caution**



---

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

---

---

## Support services

Applied Microsystems provides a full range of support services. NetROM is covered by a 90-day warranty. Additional support agreements are available 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 ASI number available.

### Telephone

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

(425) 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](mailto:support@amc.com)

If you have product suggestions or requests, use the following address:

[netrom@amc.com](mailto:netrom@amc.com)

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

<http://www.amc.com>

See the Applied web page for NetROM application notes.

### FAX

(425) 883-3049



## Chapter 2

# NetROM Services

---

This chapter provides a general description of NetROM's features.

<b>Contents</b>	<b>Page</b>
"NetROM console"	2-2
Communications paths	2-3
Debug path	2-4
Console path	2-7
Channel paths	2-9
Download path	2-11
Optional downloadable RAM module	2-16
Command and status signals	2-16
NetROM LEDs	2-18



---

## NetROM console

To set up NetROM and establish communications between NetROM and host, there is one communication path from the host to the NetROM console. The NetROM console processes communications to NetROM from several input connections:

- ❑ Console serial port (for user-to-NetROM communications)
- ❑ TELNET port (for user-to-NetROM communications)
- ❑ NetROM console port (for debugger-to-NetROM communications)
- ❑ NetROM debug control port (for debugger-to-NetROM communications)

The console serial port serves several purposes.

- ❑ It allows access to 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.
- ❑ 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 manually. 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 7 for information on alternate NetROM interfaces.)
- ❑ You may occasionally want to monitor traffic between NetROM and the target system on the console path, the debug path, and/or one of the channel paths. If enabled, data passed between NetROM and target can be echoed to the NetROM console; see the *set consecho*, *set debugecho*, and *set chanecho* commands for details. This is particularly useful when integrating NetROM with various debugger/monitor applications.

- Displays 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.

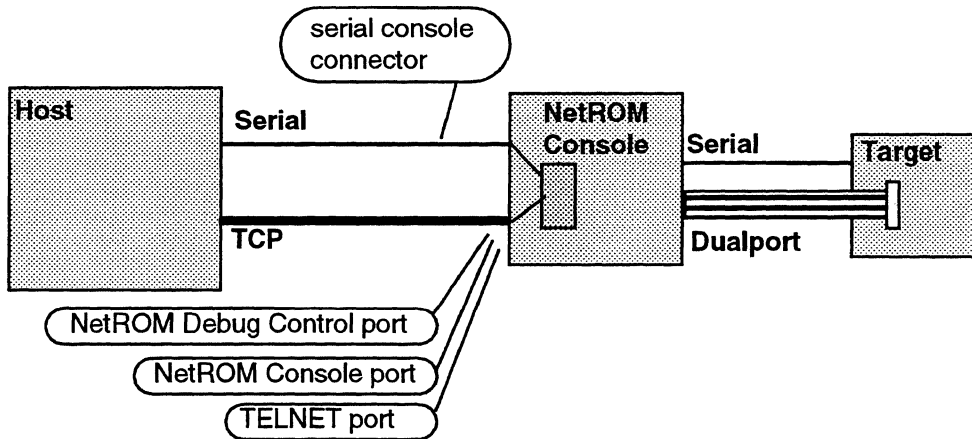


Figure 2-1 NetROM console communications channels

## Communications paths

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

### Debug path

How a debugger running on a host computer communicates with a software monitor on the target system. This allows you 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.

## **Console path/channel path**

How the user interacts with the target system; i.e., configure, control, and monitor. This can consist of a serial port running a simple terminal emulator that processes commands or a TELNET session on a direct TCP link to a NetROM port. These paths allow the user to easily inspect and monitor the system for bugs or unexpected behavior.

## **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.

The following sections provide detailed information on each path.

---

## **Debug path**

Many development environments for embedded systems use symbolic debuggers. These debuggers run on the host system but communicate with the target system, usually 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 is generally a software monitor which perform the operations requested by the host side.

The communications methods from NetROM to target and from host to NetROM are independent of each other. 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.

### NetROM-to-target communication

The debug path uses either serial or “dualport” (emulation memory mailbox) channels to communicate with the target system.

The dualport 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 detail in Chapter 9.

Note



---

There is a single serial channel between NetROM and target. Communication on the serial channel will be seen by any path using the channel.

---

Note



---

A single dualport channel is shared between the console path and the debug path. If you want to reserve the dualport channel for use by the debug path, consider using a serial channel for the console path or use one of three channel paths for NetROM-to-target communications.

---

### Host-to-NetROM communication

There are two paths between NetROM and the debugger. Both are TCP connections. One is the *debug data path*; data received from the debugger is passed directly to the target. The other is the *debug control path*; control data received from the debugger

is passed directly to NetROM, which allows debuggers to perform such tasks as directly reset the target or download a new image.

TELNET is the recommended method for TCP communications with NetROM. (See Chapter 8 for information on alternate NetROM interfaces.) Using TELNET for all types of communication with the target means that you 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 you can communicate with the target — via NetROM — from anywhere in the office or lab which provides TELNET.

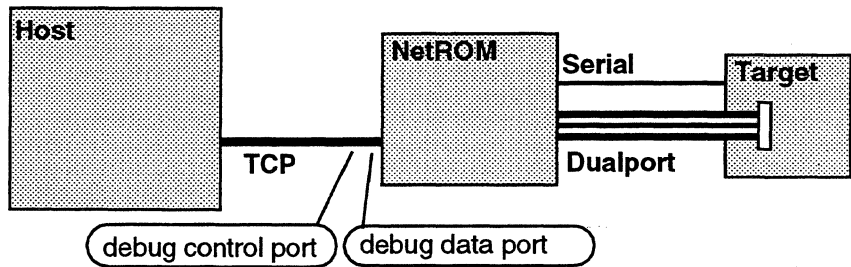


Figure 2-2 Debug path communications channels

---

## 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 you must rely on LED displays and DIP switches or similar methods to determine what the target is doing and to give it commands. NetROM provides tools to communicate directly with any of these types of target systems.

The communications methods from NetROM to target and from host to NetROM are independent of each other.

### NetROM-to-target communication

The console path uses either serial or "dualport" (emulation memory mailbox) channels to communicate with the target system.

The target doesn't need to have a serial port in order to provide a console to the user. By using the emulation memory, which is shared between NetROM and the target, to pass messages, the target can communicate directly with the user during debugging. Using emulation memory to pass messages 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 user.

The dualport 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 detail in Chapter 9.

Note



---

There is a single serial channel between NetROM and target. Communication on the serial channel will be seen by any path using the channel.

---

Note



---

A single dualport channel is shared between the console path and the debug path. If you want to reserve the dualport channel for use by the debug path, consider using a serial channel for the console path or use one of three channel paths for NetROM-to-target communications.

---

### Host-to-NetROM communication

Use TCP or serial communications between host and NetROM for the console path. Once communication is established between host and NetROM, use the *tgtcons* command (page 5-29) to establish a console session.

TELNET is the recommended method for TCP communications with NetROM. (See Chapter 8 for information on alternate NetROM interfaces.) Using TELNET for all types of communication means that you 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 you can communicate with the target — via NetROM — from anywhere in the office or lab which provides TELNET.

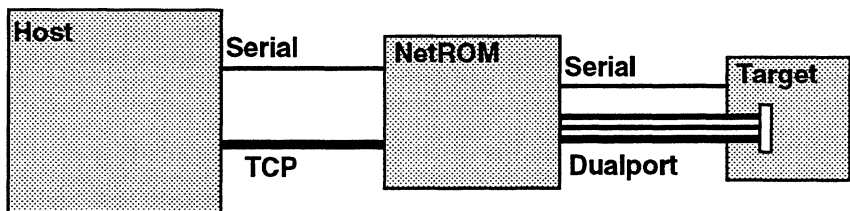


Figure 2-3 Console path communications channels

---

## Channel paths

The three channel paths provide alternate communication mechanisms between the user and the target.

Because the debug path and console path share the same dualport channel, you may want to use one of the channel paths to communicate with the target.

The communications methods from NetROM to target and from host to NetROM are independent of each other.

### NetROM-to-target communication

The channel paths use either serial or “dualport” (emulation memory mailbox) channels to communicate with the target system.

Three independent channels are available with dualport communications. The dualport 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 detail in Chapter 9. In addition, you can use buffer-oriented I/O. See “nr\_GetMsg” on page 9-16 and “nr\_PutMsg” on page 9-19.

Note



---

There is a single serial channel between NetROM and target. Communication on the serial channel will be seen by any path using the channel.

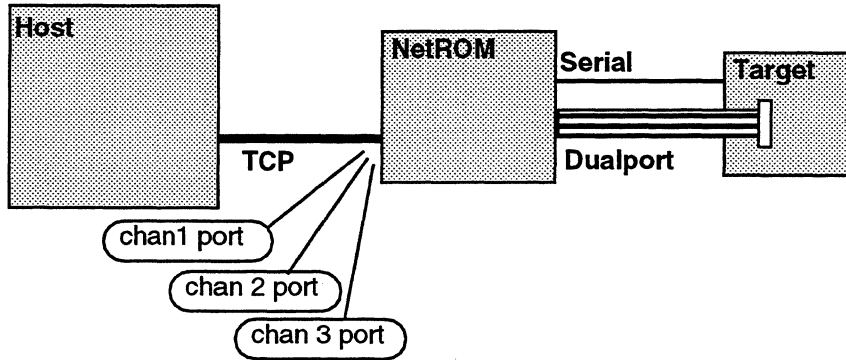
---

### Host-to-NetROM communication

TELNET is the recommended method for TCP communications with NetROM. (See Chapter 8 for information on alternate NetROM interfaces.) Using TELNET for all types of



communication with the target means that you 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 you can communicate with the target — via NetROM — from anywhere in the office or lab which provides TELNET.



**Figure 2-4** Channel path communications channels

---

## 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.

## Emulation memory

NetROM has 1 MB or 4 MB of emulation SRAM. It consists of four partitions, where the size of each partition is one-fourth of the maximum, or 256K for 1 MB NetROMs and 1 MB for 4 MB NetROMs. These partitions of emulation memory connect to the target system using ribbon cables which end in a header or connector used for the particular type of ROM being emulated.

The partitions of emulation memory will be referred to as "emulation pods". The ribbon cables and ROM connectors will be referred to as "emulation cables", of which there are two types - active and passive. The ROM connectors will be referred to as "plugs".

In general, emulation memory can be used to support a single ROM, multiple ROMs used to extend the memory space, and multiple ROMs used to create multiple-byte words.

With passive emulation cables, each emulation pod can emulate a single 8-bit ROM. The emulation pods can also be combined to support larger or wider ROMs.

With active emulation cables, emulation pods are used in pairs and can emulate a single 8-bit ROM, or a single 16-bit ROM. The emulation pod pairs can also be combined to support larger or wider ROMs. When referring to the use of active emulation cables, pod pairs will simply be referred to as pods.

## ROM groups

When multiple ROMs are in a parallel or serial organization, they become a ROM group for emulation purposes. A parallel organization is when ROMs are used to create words wider than a single ROM. A serial organization is when ROMs are used to a memory space longer than a single ROM.

Serial and parallel organizations are not mutually exclusive; a target with four 8-bit ROMs might use them to create twice as many 16-bit words as it could with only two.

To emulate a ROM group, you will need to define several attributes. If your target has multiple ROM groups that you wish to emulate, you can define unique groups for each using “set rgconfig” on page 5-44. To select the default ROM group upon which other commands will act, use “romgroup” on page 5-135.

### ROM group defining attributes

ROM groups have five defining attributes:

1. ROM type

All ROMs in a ROM group are the same type of ROM. Thus, it is not necessary to specify the ROM type for each ROM in a ROM group, but only for the ROM group as a whole. See “romtype” on page 5-136.

2. Word size

This specifies the size of the word of the ROM group.

For example, two 27c020 ROMs are used in parallel to create a ROM space with 256 Kwords. Each word is 16 bits, not eight bits, wide.

Word size is closely tied to ROM count, described below. See “wordsize” on page 5-142. The target will generally have an address which is the start of ROM space and will read whole words from that address.

3. ROM count

This is the total number of emulated ROMs.

If the ROM count is greater than the word size divided by 8, then some of the emulated ROMs are being used serially to create a ROM address space longer than is possible with a single set of emulated ROMs operating in parallel. See “romcount” on page 5-134.

For example, four 27c020 ROMs use a word size of 16 bits. In this case, the word size (16) divided by 8 is 2, so the ROMs 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.

Table 2-1 and Table 2-2 show the valid combinations of ROM count and word size. Valid combinations are dependent upon the type of emulation cables being used and the type of ROMs being emulated.

**Table 2-1** Combinations of ROM count and word size (*passive* cables)

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

**Table 2-2** Combinations of ROM count and word size (*active* cables)

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

#### 4. Pod order

For target systems which use multiple 8-bit ROMs, in serial or in parallel, you can specify the correspondence between emulation pods and ROM bytes.

For example, a 16-bit data bus target might number its sockets 0 and 1, according to where the byte is within the 16-bit word. You might want pod 0 to be byte 0 on the target and pod 1 to be byte 1, or might want pod 1 to be byte 0 and pod 0 to be byte 1.

See “podorder” on page 5-127. Note, “podorder” and “wordsize”/“romcount” provide the same information in a different form.

#### 5. Writability

This allows the target to set breakpoints in a ROM image, or to modify the image in other ways. ROM 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 “Dualport protocol” on page F-10.

The emulation pod memory can be configured to emulate either flash ROM or static RAM. The difference is how NetROM reacts to the WR signal when OE is asserted. The environment variable “writemode” controls this attribute. The default mode is FLASH. See “writemode” on page 5-143.

### **Other ROM group attributes**

In addition, ROM groups have two other attributes:

#### 1. Group name

ROM groups can be assigned a name, so that you will not have to remember which ROM group is being used for what purpose. Names can be any mnemonic, and are optional. See “set rgroupname” on page 5-47.

## 2. Target address

This is the 32-bit address indicating the start of the ROM group in the target's address space. The ability to specify a target-side starting address allows you 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. See “groupaddr” on page 5-122.

## Configuring ROM groups

If you want to emulate only one ROM group, configuring it is quite simple. As described above, you specify the ROM type, the word size, and the ROM count for the default ROM group (group 0) simply by stating your preferences on the command line. ROM group configuration can be done as part of a startup batch file. After the ROM group has been configured, the group can be loaded with an image and emulation can begin.

If you want to emulate multiple ROM groups, you can define unique groups for each using “set rgconfig” on page 5-44. To select the default ROM group upon which other commands will act, use “romgroup” on page 5-135.

## Downloading ROM groups

Each ROM group can be downloaded independently. NetROM takes advantage of LAN speeds to accomplish fast downloads. Downloads can be accomplished in several ways:

- Trivial File Transfer Protocol (TFTP), a standard Internet file transfer program.
- TCP connections to a port on NetROM.

---

## Optional downloadable RAM module

Applied Microsystems' optional downloadable RAM module is a licensed applications package that adds functionality to NetROM. The Virtual Ethernet module gives target systems the ability to become Ethernet communications devices without requiring that they have Ethernet hardware.

The application is licensed to individual NetROM units and is linked to a specific NetROM version. That is, when you purchase a 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 diskette contains 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 setup and usage instructions for the Virtual Ethernet module is in Chapter 11.

---

## Command and status signals

### Command signals

NetROM provides a set of eight command signals which can be mapped to arbitrary traces on the target system. The command signals are "active low," which means when asserted (set to "on") the signal is near ground potential.

There are certain electrical considerations involved in using the command signals.

- ❑ Command signals are tri-state; that is, when they are not asserted, they are “not connected” to the target.
- ❑ 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 meant to be used in either open-collector circuits or circuits which drive small amounts of current. 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 current-limiting resistor between the command signal and the target trace.

### Status signals

NetROM also provides a set of status signals which 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 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 you can 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 so that the LED lights up when the signal is high. See “ledmap” on page 5-92 and “di ledmap” on page 5-64.

### Defaults

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, you can apply a 100-millisecond hardware reset to the target remotely; refer to “reset” on page 5-96.



---

# NetROM LEDs

## Main unit

### Front panel

The front of NetROM has two sets of LEDs, with one set for each cable and one LED for each emulation pod. When you plug the cable correctly into the target, the corresponding NetROM green LED lights. If the cable is plugged in backwards, the LED will not light.

### Back panel

The back of 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 2-5 shows the 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. See "ledmap" on page 5-92 and "di ledmap" on page 5-64. The single red LED to the left indicates that power is on.

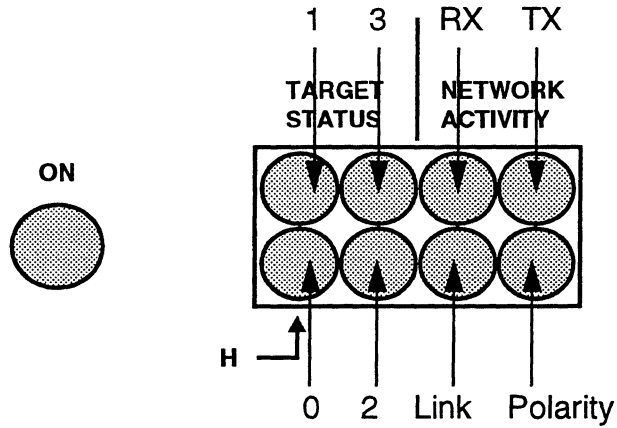


Figure 2-5 NetROM LEDs

### Active cables

The active cable includes two LEDs, indicating read and write cycles.

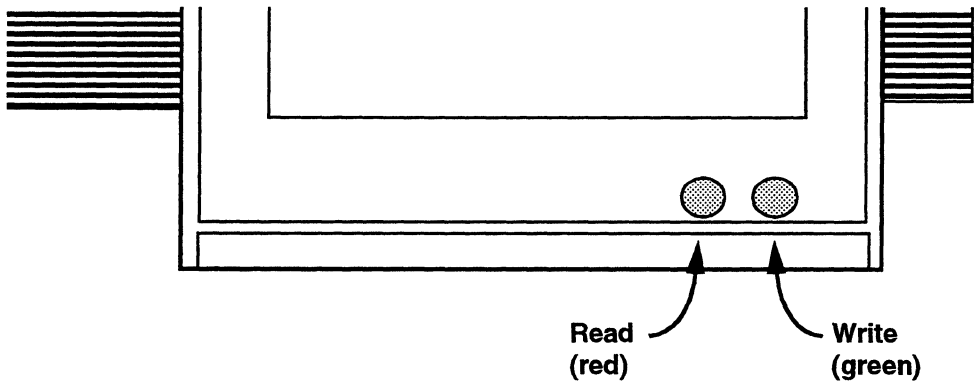


Figure 2-6 Active cable LEDs



## Chapter 3

# Hardware Installation

---

This chapter provides instructions for installing NetROM hardware.

<b>Contents</b>	<b>Page</b>
Collecting equipment	3-2
Connecting power	3-5
Connecting to Ethernet	3-6
Connecting ROM emulation cables	3-8
Connecting NetROM console serial port	3-27
Connecting target serial port	3-27
Connecting the write signal	3-29
Connecting the reset signal	3-31

**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.

---

---

## Collecting equipment

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

- An AC power cord and transformer provided with your NetROM.
- The ROM emulation cables you ordered.
- A means of connecting to the Ethernet. Ethernet connection may be accomplished two ways:
  - Via a twisted pair MAU
  - 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.

The following sections explain how to install external connections to the various connectors on NetROM. Figure 3-2 illustrates the front and rear panels of the NetROM.

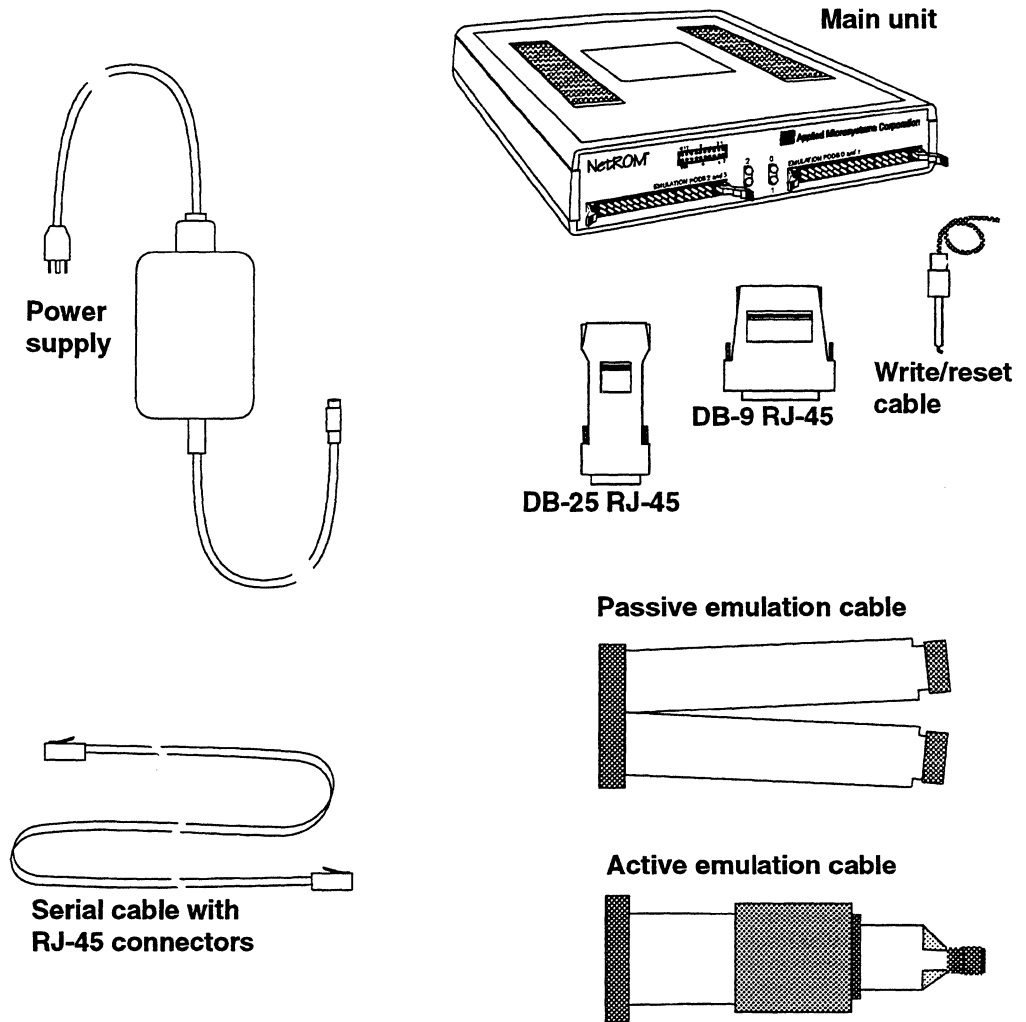
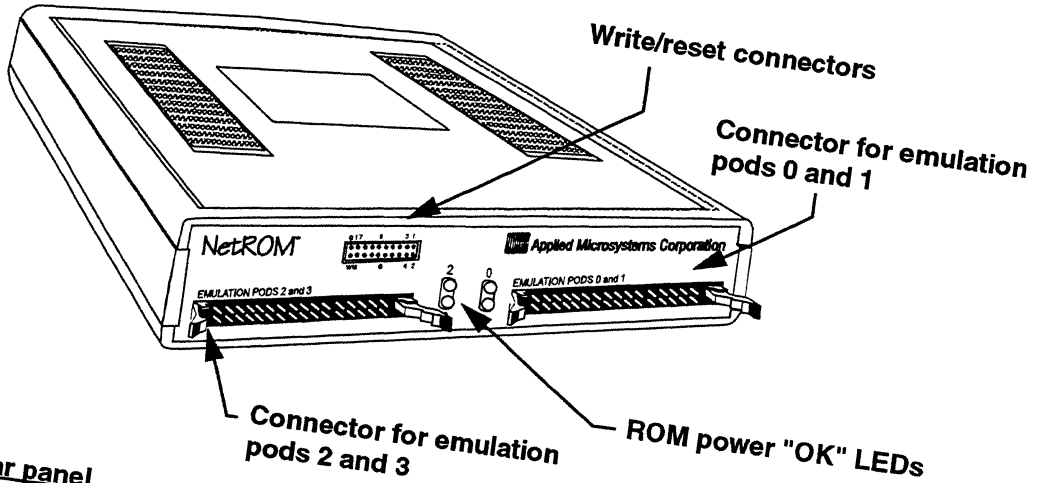


Figure 3-1 NetROM and accessories

**Front panel**



**Rear panel**

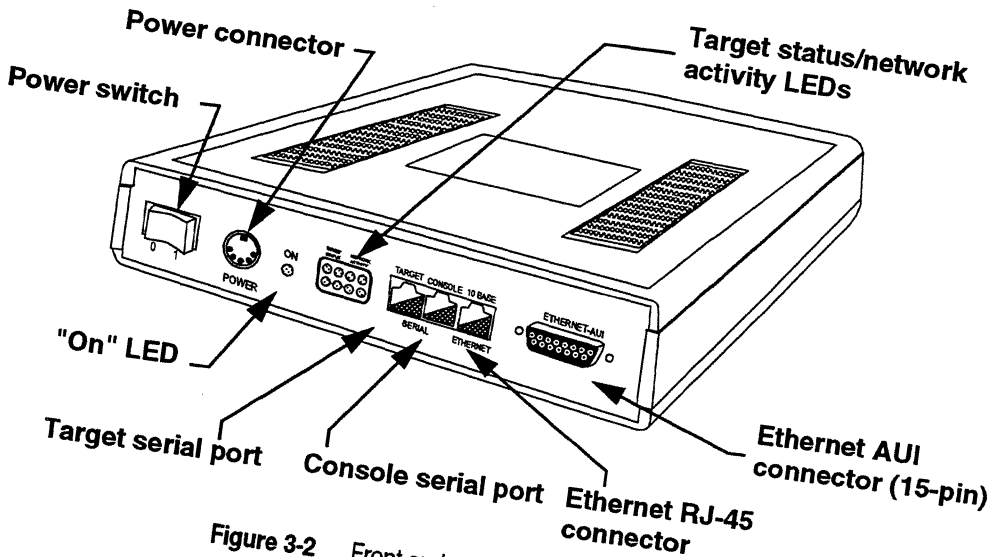


Figure 3-2 Front and rear panel connections

---

## Connecting power

### Connecting AC power cord

**Caution**



---

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

---

➤ **To connect AC power:**

1. Make certain the power switch on NetROM is turned off.
2. Attach the power cord to the connector labeled POWER on the rear panel of NetROM.
3. Plug the other end of the power cord into a grounded AC wall outlet.



---

## Connecting to Ethernet

NetROM can be connected to the Ethernet via one of two means. Figure 3-2 on page 3-4 shows the location of the connectors on the back of the NetROM, and Figure 3-3 on page 3-7 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.

An Ethernet transceiver is required when connecting NetROM to either thin or thick Ethernet cable.

➤ **To use an Ethernet Transceiver:**

- Connect the transceiver 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.

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.

---

➤ **To use an RJ-45 Connector:**

- Connect to 10 Base-T networks via the RJ-45 connector marked ETHERNET (Figure 3-3).

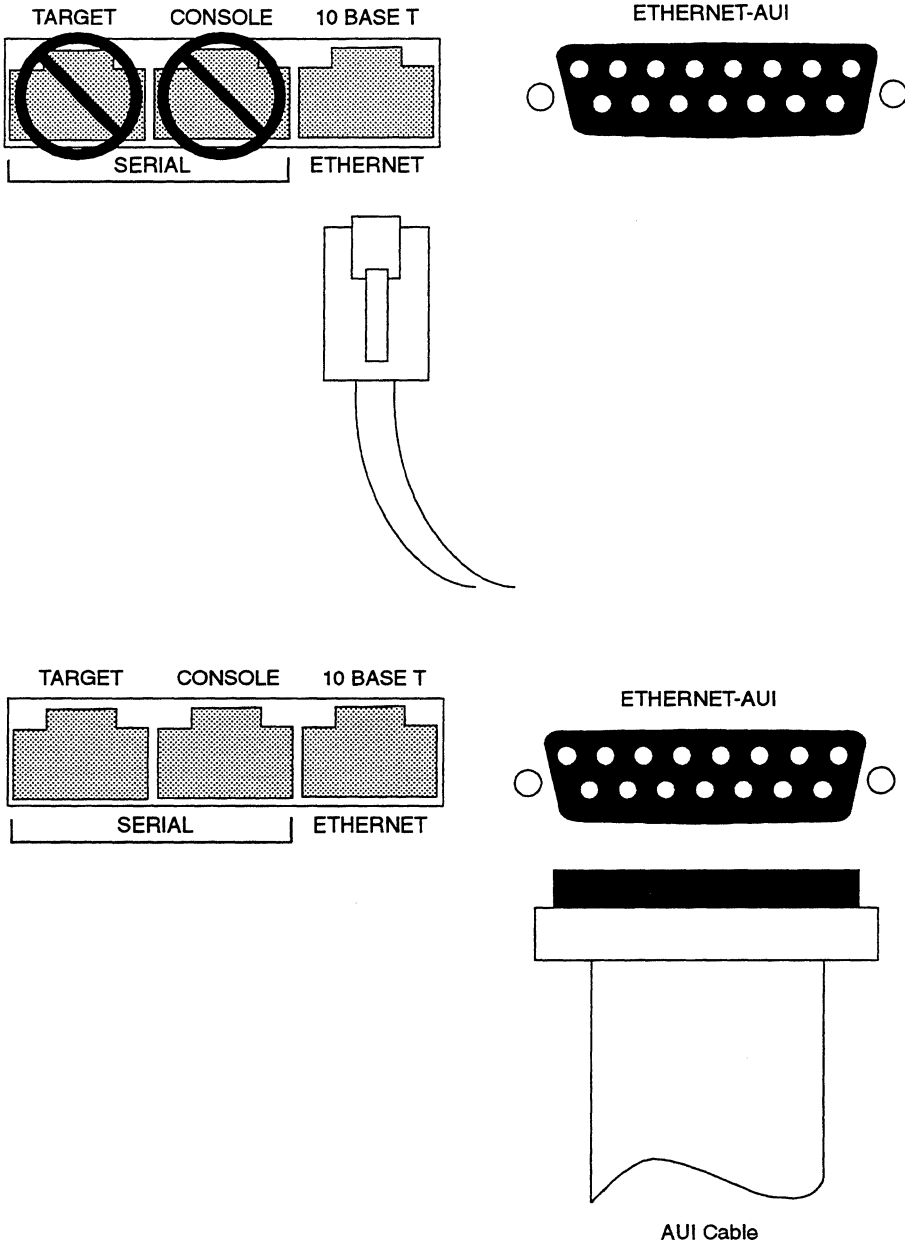


Figure 3-3 Connecting Ethernet

---

# Connecting ROM emulation cables

## Cable configurations

Because of the extensive list of supported ROMs, several configurations of cable assemblies are provided:

- Active emulation cables
- Passive emulation cables
- Passive emulation cables with the "4 to 5" cable converter

**Active cables** contain active circuitry that is software configured for the type of memory being emulated.

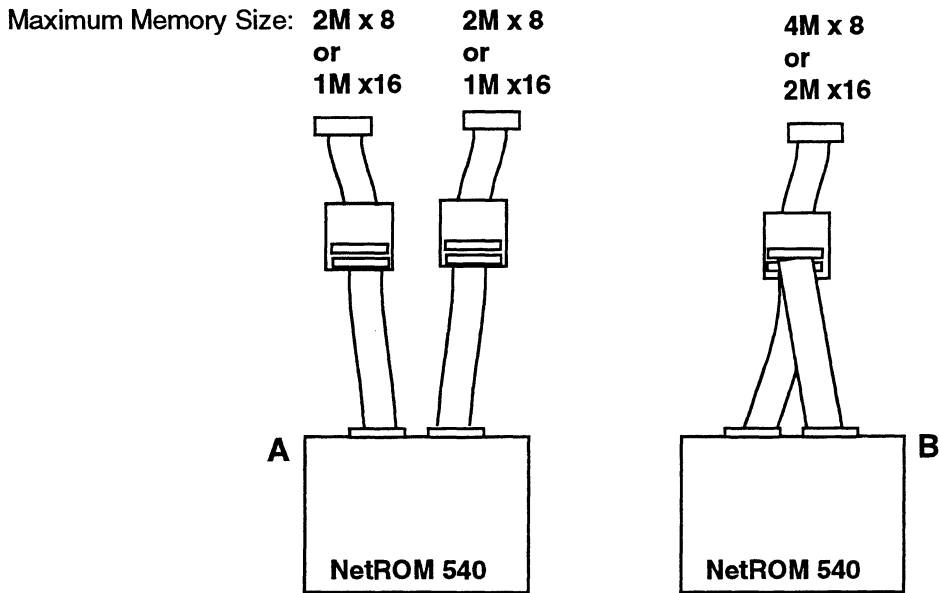
**Passive cables** originally designed for use with NetROM 400 series can be used with NetROM 500 series by connecting them through the cable converter.

Examples of supported cable configurations are shown in Figure 3-4, Figure 3-5, and Figure 3-6.

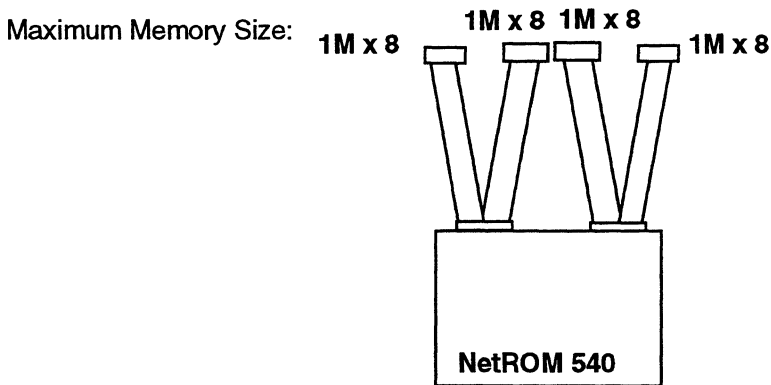
### Assembly of ROM emulation cables

Depending on the device you are emulating you may need to connect several assemblies to create your final cable assembly configuration, prior to connecting to your target (see "Connecting ROM emulation cables to target" on page 3-23).

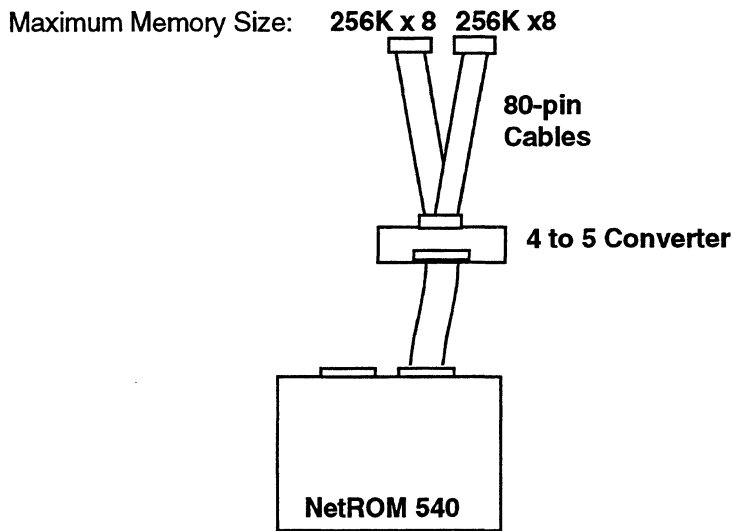
Examples of cable assemblies are shown in Figure 3-7 and Figure 3-8. All connectors are keyed.



**Figure 3-4** Active emulation cable configurations



**Figure 3-5** Passive emulation cable configuration



**Figure 3-6** Passive emulation cable configuration

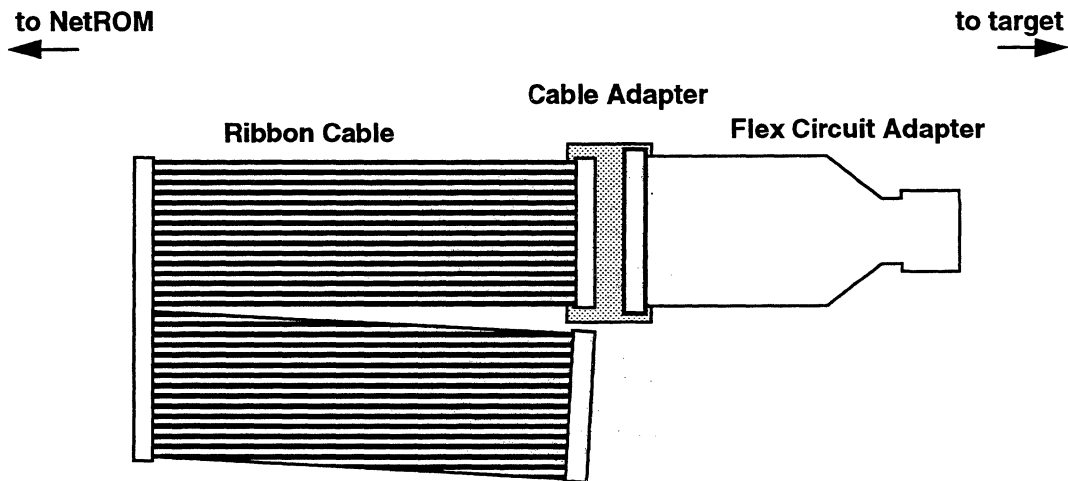


Figure 3-7 Cable assembly example

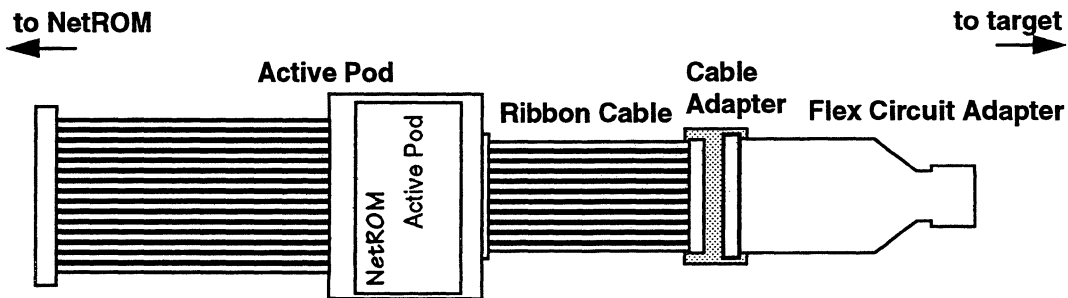


Figure 3-8 Cable assembly example

## Connection steps

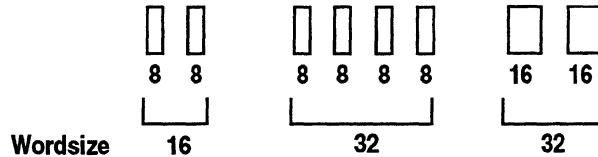
- **To connect to a target system:**
  1. Identify the type and number of ROMs to be emulated. An emulation cable is required for each ROM.
  2. For more than one ROM, determine whether they are in a parallel and/or serial configuration on the target.
  3. Determine which emulation pods should be used.
  4. Connect ROM emulation cables to NetROM
  5. Connect ROM emulation cables to target.

The following topics are covered in this section:

<b>Section</b>	<b>Page</b>
Parallel or serial?	3-13
Which emulation pods should be used?	3-14
Connecting ROM emulation cables to NetROM	3-22
Connecting ROM emulation cables to target	3-23

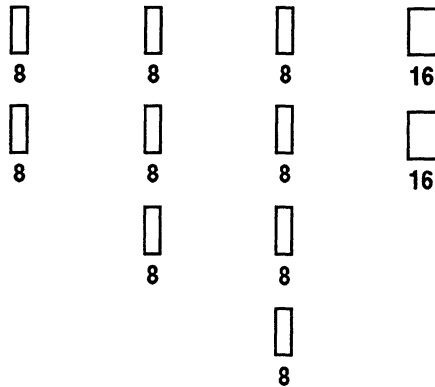
### Parallel or serial?

**Parallel.** If the target system has a multiple-byte word size and more than one ROM is used to create this word, the ROMs are parallel and a parallel emulation setup is used.



**Figure 3-9** Possible parallel configurations

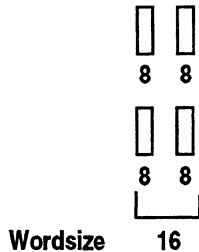
**Serial.** If a target uses additional ROMs to provide more memory space than is supplied in a single ROM the ROMs are serial and a serial emulation setup is used.



**Figure 3-10** Possible serial configurations



**Parallel and serial.** If the target uses ROMs in a parallel/serial combination a parallel/serial emulation setup is used.



**Figure 3-11** Parallel/serial configuration

### **Which emulation pods should be used?**

With 4 MB NetROM, passive cables support 8-bit ROMs up to 1 MB; active cables support 8-bit ROMs from 1 MB-4 MB and 16-bit ROMs.

With 1 MB NetROM, passive cables support 8-bit ROMs up to 256K; active cables support 8-bit ROMs from 256K to 1 M and 16-bit ROMs

NetROM software has a default mapping of emulation pods to target byte ordering. This mapping can be overridden using the "podorder" environment variable discussed on page 5-127. You need to ensure that the emulation cables connect the ROMs to the emulation pods in the correct order.

The following tables illustrate configurations for ROMs used for 8, 16, or 32-bit words in parallel, serial, or both configurations., using passive and active pods with 4 MB NetROM. The default mapping and big-endian byte ordering are assumed. Use the tables to determine which emulation pod should be used to emulate each ROM.

**8-bit word size/passive cables.** There are four possible configurations: using 1, 2, 3 or 4 ROMs. Table 3-1 shows mapping between NetROM emulation pods and the target's ROM addresses. Also included are the data bits supplied by each emulation pod.

**Table 3-1** Emulation of an 8-bit word size with **passive** emulation cables

Pod	Number of ROMs Emulated				Data Bits
	1	2	3	4	
0	bytes 0 to (1M-1) 0x00000 to 0xfffff	bytes 0 to (1M-1) 0x00000 to 0xfffff	bytes 0 to (1M-1) 0x00000 to 0xfffff	bytes 0 to (1M-1) 0x00000 to 0xfffff	D0-D7
1	N/A	bytes 1M to (2M-1) 0x100000 to 0x1fffff	bytes 1M to (2M-1) 0x100000 to 0x1fffff	bytes 1M to (2M-1) 0x100000 to 0x1fffff	D0-D7
2	N/A	N/A	bytes 2M to (3M-1) 0x200000 to 0x2fffff	bytes 2M to (3M-1) 0x200000 to 0x2fffff	D0-D7
3	N/A	N/A	N/A	bytes 3M to (4M-1) 0x300000 to 0x3fffff	D0-D7

**8-bit word size/active cables.** There are two possible configurations: using 1 or 2 ROMs. Table 3-2 shows mapping between NetROM emulation pods and the target's ROM addresses. Also included are the data bits supplied by each emulation pod.

**Table 3-2** Emulation of an 8-bit word size with active emulation cables

<b>Number of ROMs Emulated</b>				
<b>Configuration A in Figure 3-4</b>		<b>Configuration B in Figure 3-4</b>		
<b>Pod</b>	<b>1</b>	<b>2</b>	<b>1</b>	<b>Data Bits</b>
0	bytes 0 to (2 M-1) 0x00000 to 0x1ffff	bytes 0 to (2 M-1) 0x00000 to 0x1ffff	bytes 0 to (4 M-1) 0x00000 to 0x3ffff	D0-D7
2	N/A	bytes 2 M to (2 M-1) 0x200000 to 0x3ffff		D0-D7

**16-bit word size/passive cables.** There are two possible configurations: using 2 or 4 ROMs. Table 3-3 shows the mapping between NetROM emulation pods and the target's ROM addresses. Also included are the data bits supplied by each emulation pod.

**Table 3-3** Emulation of a 16-bit word size with **passive** emulation cables

Pod	Number of ROMs Emulated				Data Bits
	1	2	3	4	
0	N/A	even bytes 0 to (2M-1) 0x0 to 0x1fffff	N/A	even bytes 0 to (2M-1) 0x0 to 0x1fffff	D0-D7
1	N/A	odd bytes 0 to (2M-1) 0x0 to 0x1fffff	N/A	odd bytes 0 to (2M-1) 0x0 to 0x1fffff	D8-D15
2	N/A	N/A	N/A	even bytes 2M to (4M-1) 0x200000 to 0x3fffff	D0-D7
3	N/A	N/A	N/A	odd bytes 2M to (4M-1) 0x200000 to 0x3fffff	D8-D15

**16-bit word size/active cables.** There are two possible configurations: using 1 or 2 ROMs. Table 3-4 shows the mapping between NetROM emulation pods and the target's ROM addresses. Also included are the data bits supplied by each emulation pod.

**Table 3-4** Emulation of an 16-bit word size with **active** emulation cables

<b>Number of ROMs Emulated</b>					
		<b>Configuration A in Figure 3-4</b>		<b>Configuration B in Figure 3-4</b>	
<b>Pod</b>	<b>1</b>	<b>2</b>	<b>1</b>		<b>Data Bits</b>
0	bytes 0 to (2M-1) 0x0 to 0x1ffff	bytes 0 to (2 M-1) 0x0 to 0x1ffff	bytes 0 to (4M-1) 0x0 to 0x3ffff		D0-D15
2		bytes 2 M to (4 M-1) 0x200000 to 0x3ffff			D0-D15

**32-bit word size/passive cables.** There is one possible configuration: using four ROMs. Table 3-5 shows mapping between NetROM emulation pods and the target's ROM addresses. Also included are the data bits supplied by each emulation pod.

**Table 3-5** Emulation of a 32-bit word size with **passive** emulation cables

Pod	Number of ROMs Emulated				Data Bits
	1	2	3	4	
0	N/A	N/A	N/A	byte zero, 0 to (4M-1) 0x0 to 0x3ffff	D0-D7
1	N/A	N/A	N/A	byte one, 0 to (4M-1) 0x0 to 0x3ffff	D8-D15
2	N/A	N/A	N/A	byte two, 0 to (4M-1) 0x0 to 0x3ffff	D16-D23
3	N/A	N/A	N/A	byte three, 0 to (4M-1), 0x0 to 0x3ffff	D24-D31

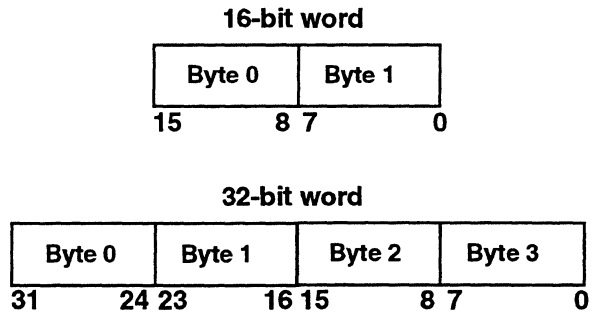
**32-bit word size/active cables.** There is one possible configuration: using two ROMs. Table 3-6 shows mapping between NetROM emulation pods and the target's ROM addresses. Also included are the data bits supplied by each emulation pod.

**Table 3-6** Emulation of an 32-bit word size with **active** emulation cables

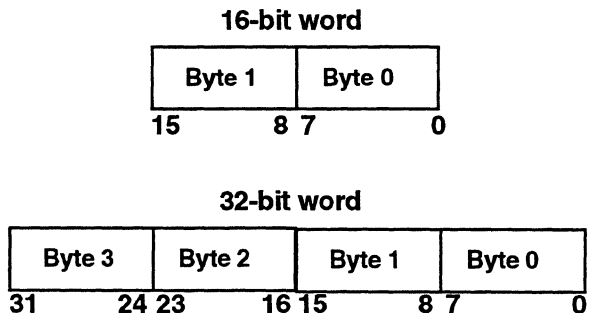
<b>Number of ROMs Emulated</b>		
<b>Configuration A in Figure 3-4</b>		
<b>Pod</b>	<b>2</b>	<b>Data Bits</b>
0	byte zero/one    bytes 0 to (4M-1) 0x0 to 0x3ffff	D0-D15
2	byte two/three    bytes 0 to (4M-1) 0x0 to 0x3ffff	D16-31

**Byte ordering.** 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 cable to plug into the target system's ROM sockets.

### Big Endian Byte Ordering



### Little Endian Byte Ordering



**Figure 3-12** Big-endian/little-endian byte ordering

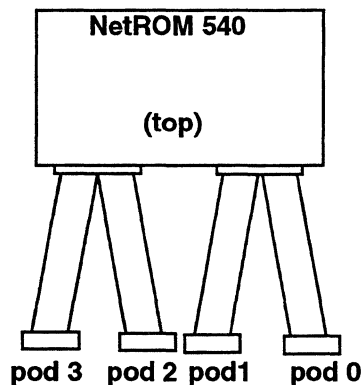


## Connecting ROM emulation cables to NetROM

The ROM emulation cables connect to NetROM through 100-pin female cable connectors. The configurations vary depending on the type of emulation cables you are using and the size and format of the memory you are emulating. The cables vary depending on the format of the ROM package located on the target system.

Active cables can connect to one ROM, so you need to consider only whether to connect to pods 0 and 1 or pods 2 and 3.

Passive cables can connect to two ROMs, but terminate in one connector that plugs into NetROM. You need to know which cable connects to which emulation pod. See the following figure for the cable/pod association.



**Figure 3-13** Pod/cable association

If you are using the 4 to 5 converter, first connect the target-end cables to the converter, then determine whether to connect to pods 0 and 1 or pods 2 and 3.

- **To use emulation pods 0 and 1**
  - Connect the emulation cables to the connector labeled EMULATOR PODS 0 AND 1.

➤ **To use emulation pods 2 and 3**

- Connect the emulation cables to the connector labeled EMULATOR PODS 2 AND 3.

See Figure 3-2 and “podorder” on page 5-127.

**Connecting ROM emulation cables to target**

Instructions follow for connecting DIP, PLCC, header, and surface mount style cables to your target.

Caution



---

NetROM and the target contain components that are subject to damage from electrostatic discharge. Whenever you are handling 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 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.

Caution



---

Plugging in ROM cables improperly may damage the NetROM.

---

➤ **To connect DIP cables:**

1. Turn off power to the target system.
2. Remove any ROMs currently in the ROM sockets on the target.
3. Align pin 1 on the ROM emulation cable DIP connector with pin 1 on the target DIP socket and insert.

The ROM power “OK” LEDs (see Figure 3-2 on page 3-4) will light when the cables are plugged in correctly and the target system is powered on.

### **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.

**Caution**



---

Plugging in ROM cables improperly may damage the NetROM.

---

➤ **To connect PLCC cables:**

1. Turn off power to the target system.
2. Remove any ROMs currently in the target sockets.
3. Align pin 1 on the emulation cable to pin 1 on the target PLCC socket and insert; one corner of the emulation cable's PLCC plug is cut off to indicate the location of pin 1.

The power “OK” LEDs (see Figure 3-2 on page 3-4) will light when the cables are plugged in correctly and the target system is powered on.

### **Connecting header style cables**

Header-style cables are used to connect NetROM to targets with a 60-pin header.

**Caution**



---

Plugging in ROM cables improperly may damage the NetROM.

---

➤ **To connect header cables:**

1. Turn off power to the target system.
2. Align pin 1 on the ROM emulation cable header connector with pin 1 on the target header socket and insert.



The ROM power “OK” LEDs (see Figure 3-2 on page 3-4) will light when the cables are plugged in correctly and the target system is powered on.

### Connecting surface mount style cables

Multiple surface mount configurations are available. Some plug in directly to a socket on your board and some require that the adapter be soldered to your target

➤ **To attach a plug-in flex adapter to the target:**

1. Turn off power to the target system.
2. Identify pin 1 on the adapter and align with pin 1 on the ROM socket. See Figure 3-14.
3. Carefully insert the adapter into the socket.

➤ **To attach a solder-down flex adapter to the target:**

Caution



---

Soldering the flex adapter to your target circuit board requires expert skill at soldering surface mount assemblies and will need to be performed under a microscope.

---

1. Turn off power to the target system.
2. On the target system, deposit additional tinning on the PCB pads. Clean the pads and apply a small amount of flux to them.
3. Identify pin 1 on the adapter and align with pin 1 on the ROM footprint. See Figure 3-14.
4. Form and test-fit the flex circuit to clear any obstacles on the target board.
5. Using a soldering iron set to about 600°F, apply heat to each pin/pad through the Kapton backing.

The ROM power “OK” LEDs (see Figure 3-2 on page 3-4) will light when the cables are plugged in correctly and the target system is powered on.

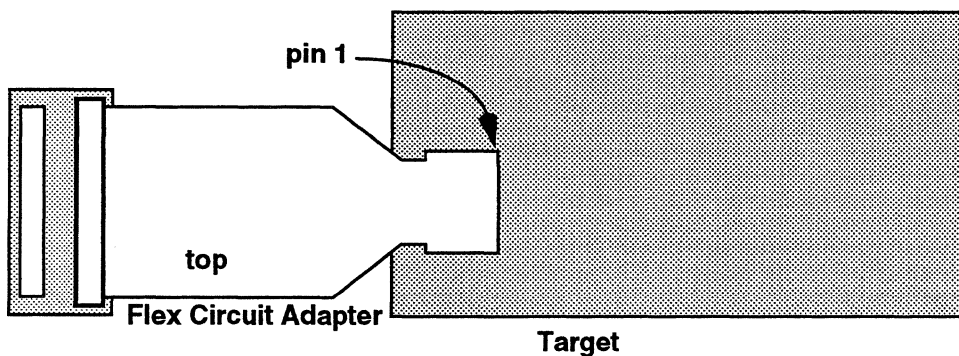
**Caution**



---

Connection between the flex circuit and the target is extremely fragile. If disturbed, the flex circuit can lift the pads from the target. Use an appropriate tape or other mechanical means to secure the flex circuit to the board.

---



**Figure 3-14** Flex circuit adapter - pin 1



---

## Connecting NetROM console serial port

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 on page 3-4 shows the location of the console port on the rear of the NetROM, and Figure 3-15 on page 3-28 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.

► **To connect to NetROM console**

- Using an RJ-45 serial cable, connect to the NetROM connector labelled **CONSOLE SERIAL** and the serial device. 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

---

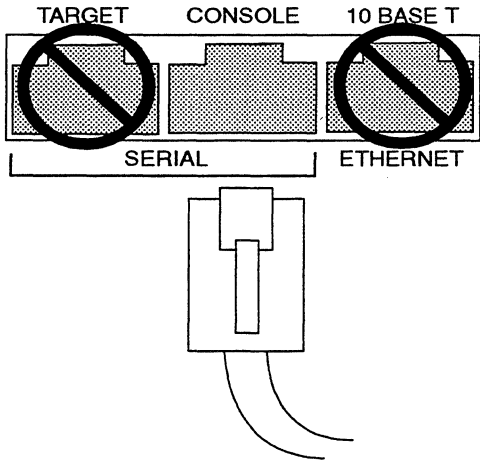
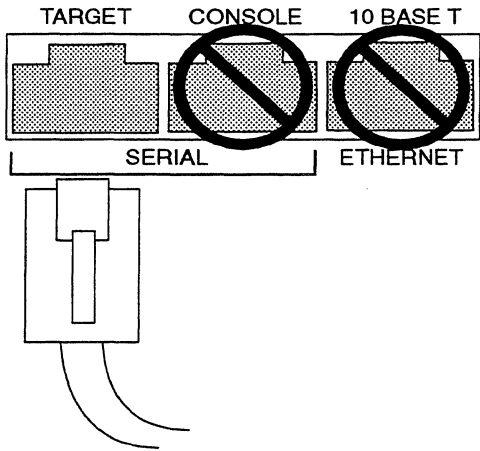
## Connecting target serial port

The NetROM target serial port allows the NetROM user to access the target system's serial port. Figure 3-2 on page 3-4 shows the location of the target serial port on the rear of the NetROM, and Figure 3-15 on page 3-28 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.

► **To connect to NetROM target serial port**

- Using an RJ-45 serial cable, connect to the NetROM connector labelled **TARGET SERIAL** and the target system's serial port.

The 9- and 25-pin connectors supplied with NetROM are DTE, not DCE. The DTR signal is always “true,” and the DSR signal is ignored.



**Figure 3-15** Connecting NetROM serial ports

---

## Connecting the write signal

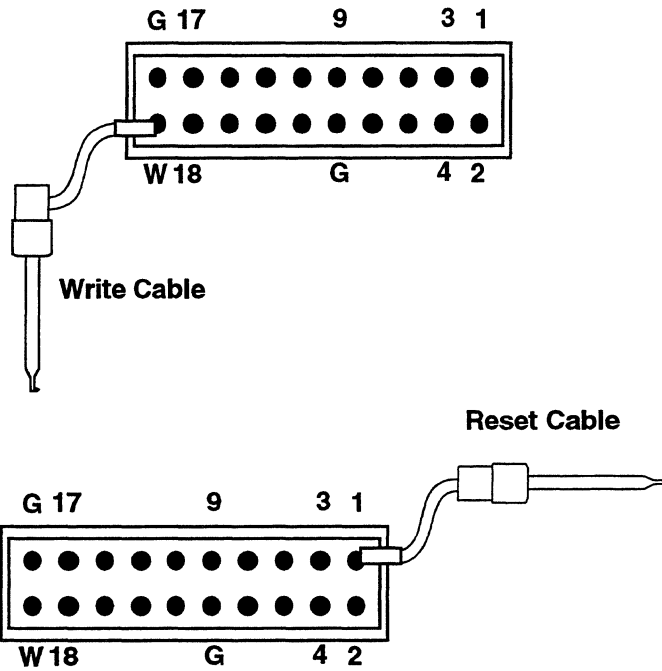
The write signal connection allows target systems to write directly to their ROM space. The data written by the target system is deposited at the specified address in NetROM's emulation memory. Writing to your ROM address space requires a write signal. There are three methods to do this:

- 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.
- Connect a write signal somewhere else on the target board.

If you need to connect the write signal, connect a jumper cable from the target system's write strobe to the write pin (pin 20) on the front panel of NetROM (see Figure 3-2 on page 3-4 and Figure 3-16 on page 3-30). The write signal is "active low" and is expected to occur in conjunction with a normal write cycle.

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.





**Figure 3-16** Connecting write and reset cables

---

## Connecting the reset signal

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! See "NetROM LEDs" on page 2-18 for more information on NetROM command signals.

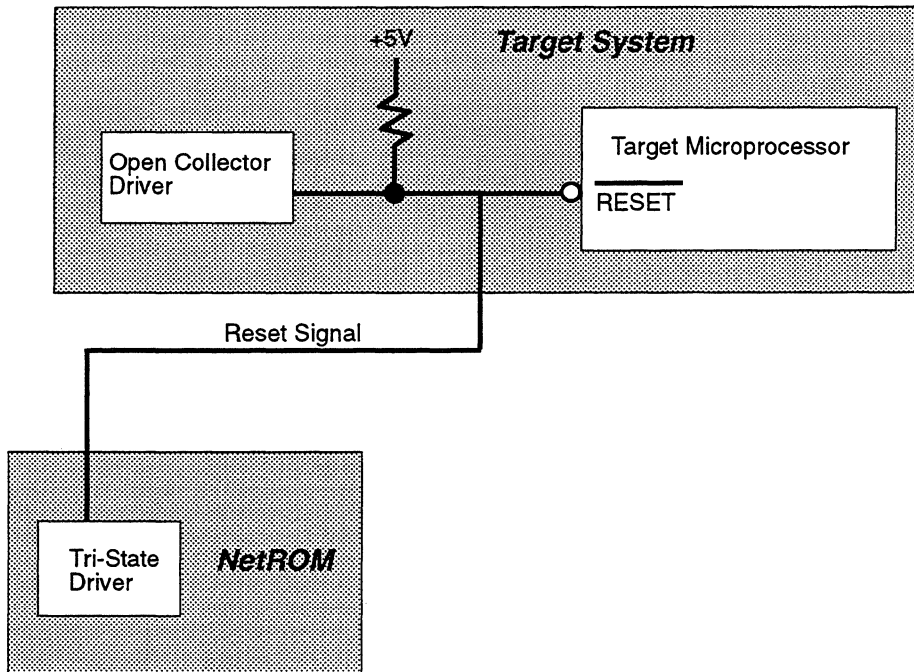
---

The reset signal is output by NetROM to reset the target system. This allows the NetROM user to reset their target system remotely.

► **To connect to NetROM reset**

- Connect a jumper cable from the target's reset signal to the Reset pin (pin 1) on the front panel of NetROM (see Figure 3-2 on page 3-4 and Figure 3-16 on page 3-30).

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 3-17.



**Figure 3-17** Connecting NetROM's reset signal

---

## Connecting the command status connector

NetROM has a special 20-pin command status connector on the front panel. It can be used as described in “Connecting the write signal” on page 3-29 and “Connecting the reset signal” on page 3-31. It can also be used to send status and control signals between NetROM and a target. See Chapter 5 in the *Hardware Interface Reference* for information on the signals and connecting to the connector.



## Chapter 4

# Software Installation

---

This chapter provides instructions for NetROM software installation and setup.

First install the software and then set up Ethernet communications. Once communications are established, create a startup batch file for your particular project.

For reference communications information, see the *NetROM Installation Notes* in the tabbed section at the end of this manual.

<b>Contents</b>	<b>Page</b>
Installing the software	4-2
Establishing network communications	4-3
Specifying NetROM startup file	4-6

---

## Installing the software

NetROM software is provided on 3 1/2 inch DOS-formatted disks. See the *NetROM Driver's and Utilities User's Guide* for additional information.

### PC

➤ **To install the software:**

1. Insert the disk labelled "NetROM, Drivers & Util" into the disk drive.
2. Run setup in the \pc\nrwin directory on the disk. For example, in Windows 3.1 select Run from the Program Manager File menu and enter:  
`a:\pc\nrwin\setup.exe`
3. Insert the disk labelled "NetROM, Firmware" into the disk drive.
4. Copy the files from the disk to your installation directory.

### Sun, HP workstations

➤ **To install the software:**

1. Insert the disk labelled "NetROM, Drivers & Util" into the disk drive on the PC.
2. Create an installation directory for the NetROM software.
3. Copy the subdirectories and files from the unix directory to the installation directory.
4. Insert the disk labelled "NetROM, Firmware" into the disk drive on the PC.
5. Copy the files from the disk to the installation directory.

## Updating the software

If this installation is an update to existing NetROM 500 series software, after completing the communications setup, load the software into NetROM. See the *set romupgrade* command on page 5-48.

---

## Establishing network communications

During normal operation, NetROM uses two network services:

- address resolution
- file transfer

Address resolution is not necessary, but it is convenient because it allows NetROM to configure its network address automatically. File transfer allows NetROM to download files into its emulation memory.

Note



---

Additional reference information is provided in the *NetROM Installation Notes* (included at the end of this manual, behind a "Notes" tab).

---

➤ **To set up communications, follow these six steps.**

1. Obtain a unique IP address for the NetROM from your system administrator.

Note



---

NetROM's Ethernet address is guaranteed to be unique. NetROM's address is printed on a label on the bottom of the unit.

---

2. Obtain the IP addresses of the Address Resolution and TFTP servers.



## Note



---

As an alternative to using RARP and BOOTP, you can configure NetROM's IP address manually. Set up a serial NetROM console session and use the *ifconfig* command (see page 5-15) to configure communications. Use the *save* command (see page 5-97) and the "bootflags" environment variable (see page 5-109) to load the IP address from non-volatile storage when NetROM is power cycled.

---

If you do not know how to identify which machine(s) on your network will do this, consult "Finding servers on your network" on page 2-14 of the *NetROM Installation Notes*. Make sure that the server software is running on the machine, and is not in a disabled state. If you are not sure how to do this, consult "Daemons on UNIX systems" on page 2-7.

### 3. Create a startup batch file.

After successfully resolving its address, NetROM will attempt to download a startup batch file. To verify that the batch file is being executed, it should do something which will be an obvious modification of NetROM's defaults. A good temporary batch file is the following:

```
begin
    set prompt -d Bob>
end
```

This sets NetROM's default prompt to be the string "Bob>"; such a prompt will be immediately obvious when you are using NetROM.

You must put the startup batch file where it can be accessed by NetROM after its address resolution. The location of the file and its name will vary depending on the "bootflags" environment variable. Using "bootflags", you can specify a name and path for your batch file, or you can use the default name and path. You can also control whether or not NetROM automatically processes your batch file when you power-cycle it. For more information on setting "bootflags", see page 5-109.



For more information on naming configuration files based on NetROM defaults, see “BOOTP” on page 2-2 or “RARP” on page 2-4 of the *NetROM Installation Notes*, and “File transfer servers” on page 2-6 of the *NetROM Installation Notes* for information on TFTP.

#### 4. Configure the servers.

Edit the address resolution server configuration files so the server is aware of NetROM. This involves editing the bootptab file if you are using BOOTP, and the ethers file if you are using RARP.

See “Address resolution servers” on page 2-1 for information on BOOTP or RARP. You will probably also want to edit the hosts file to assign a symbolic name for NetROM. See “System configuration files” on page 2-11 of the *NetROM Installation Notes* for information on system files. Note that if you are using NIS (also called the Yellow Pages) to distribute your hosts file or your ethers file you will need to update the NIS server and do a “push” operation. See “Using NIS (Yellow Pages)” on page 2-13 of the *NetROM Installation Notes* for information on NIS.

Further, if your server is running as a daemon, you may need to send it a SIGHUP to force it to re-read its configuration file. See “Daemons on UNIX systems” on page 2-7 of the *NetROM Installation Notes* for information on UNIX daemons.

#### 5. Verify that NetROM is connected to the network. See “Connecting to Ethernet” on page 3-6. If NetROM is connected to a target system, make sure that pin 1 on each emulation pod is plugged into pin 1 of the ROM socket.

#### 6. Power on NetROM.

NetROM’s LEDs should jump around for a second or two, then settle down with the heartbeat LED blinking rapidly. “NetROM LEDs” on page 2-18 describes the mapping of the LEDs.

You should now be able to log into your NetROM unit with telnet. When you TELNET to the NetROM unit, check the prompt. If the prompt is "Bob>" you are now ready to use your NetROM on a target system.

If you cannot TELNET to your NetROM unit, or if you can TELNET to the unit, but the prompt has not been set to the string specified in your startup batch file, see Chapter 3 of the *NetROM Installation Notes*.

7. After NetROM has been installed on your network, modify the startup batch file to perform whatever initialization is required for your project. Typical command sequences to include in the startup batch file set ROM type, word size, image file type, image file name, and download paths. See "Specifying NetROM startup file" on page 4-6.

See the *NetROM Installation Notes* for information on the network protocols which NetROM uses to perform address resolution and file download functions, and tips on how to configure host-side servers for your development environment.

---

## Specifying NetROM startup file

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 5 provides details on batch files and how to use them.

If you specify the *autobat* flag in the "bootflags" environment variable (see page 5-109), NetROM's address resolution mechanism at boot time will determine what startup file to process, 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.

If you set the *userbat* flag in the "bootflags" environment variable (see page 5-109), the "batchfile" (see page 5-106) and "batchpath" (see page 5-107) environment variables will determine what startup file to process.

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 5

# User Interface

---

This chapter describes the user interface to NetROM. Users will generally interact with the NetROM console via the NetROM console serial port, TELNET sessions, or direct TCP connection to the NetROM console port.

After a description of NetROM command line processing, the following command groups will be covered:

<b>Command Groups</b>	<b>Page</b>
NetROM command line processing	5-2
NetROM commands	5-9
Network interface commands	5-13
Target interface commands	5-21
Process control commands	5-35
Set commands	5-55
Display commands	5-78
ROM set commands	5-87
Miscellaneous commands	5-87
Environment variable commands	5-101

---

## 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. The most common processes are *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 5-1 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 5-1** 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 <i>ping</i> 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 console 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.

**Table 5-2** Terminal control characters

<b>Name</b>	<b>Value</b>	<b>Description</b>
eof	^D	End of file indicate that interactive input for a given command; it does not terminate a terminal session.
erase	^H	Erase the character to the left of the cursor from the input stream
intr	^C	Send a SIGINT signal to all child processes. (See also “tgtcons” on page 5-29.)
kill	^U	Erase the input line.
werase	^W	Erase the white-space-delimited “word” to the left of the cursor.

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.

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.

## History substitution

Each NetROM console session keeps track of commands it has been given. These 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 *cs* 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 5-4. 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.

## Environment variables

Environment variables affect all terminal sessions running on NetROM. All environment variables are predefined; they are primarily concerned with configuring emulation, file locations, and with establishing communications paths between the target and the host system. NetROM environment variables are summarized in Table 5-3. They are described in more detail in “Environment variable commands” on page 5-101.

**Table 5-3** Environmental variables

Variable	Description	Page
"batchfile"	Specifies default batch file to process.	5-106
"batchpath"	Sets path on the TFTP file server NetROM uses to search for batch files and RAM module locations.	5-107
"binenv"	Controls how emulation memory is written and displayed.	5-108
"bootflags"	Controls NetROM's boot-time behavior	5-109
"chanpath"	Sets channel communication path between NetROM and the target system.	5-110
"chanport"	Sets TCP/UDP port number on which NetROM accepts communications on channel path.	5-112
"consolepath"	Sets console communication path between NetROM and the target system.	5-113
"debugpath"	Sets debug communication path between NetROM and the target system.	5-115
"debugport"	Sets TCP/UDP port number on which NetROM accepts communications on debug path.	5-117
"dprbase"	Sets base address in emulation pod 0 to map dualport RAM.	5-118
"filetype"	Sets expected download file format. Supports binary, S-record, and Intel hex.	5-120

**Table 5-3** Environmental variables (Continued)

<b>Variable</b>	<b>Description</b>	<b>Page</b>
“fillpattern”	Sets byte pattern to fill emulation memory.	5-121
“groupaddr”	Sets default ROM group’s start address.	5-122
“groupwrite”	Enables or disables NetROM’s external write signal.	5-123
“host”	Sets IP address of the TFTP server used for image and batch downloads, <i>romupgrade</i> , and <i>loadmodule</i> .	5-124
“loadfile”	Sets default file to download into the default ROM group.	5-125
“loadpath”	Sets default path for downloading “loadfile” and <i>romupgrade</i> .	5-126
“podorder”	Sets pod-to-byte mapping of emulation pods in the default ROM group.	5-127
“romgroup”	Sets default ROM group.	5-135
“romcount”	Sets number of bytes in emulation as part of the default ROM group.	5-134
“romtype”	Sets ROM type being emulated by the default ROM group.	5-136
“tgtip” (optional)	Sets target machine’s IP address when Virtual Ethernet is on.	5-139
“verify”	Specifies whether downloads are verified.	5-140
“vether” (optional)	Sets Virtual Ethernet on or off.	5-141
“wordsize”	Sets size in bits of the ROM word being emulated by default ROM group.	5-142
“writemode”	Sets write mode that configures emulation memory to emulate FLASH ROM or static RAM.	5-143

## NetROM commands

Commands can be issued to the NetROM console through NetROM's console serial port, or via a network connection through a TELNET connection or to the NetROM console port.

In this section, the NetROM interface commands are grouped by functional type. Table 5-4 summarizes the commands alphabetically and gives their type and page number.

**Table 5-4** NetROM command summary

Command	Description	Type	Page
alias	Creates or deletes command "nickname".	Miscellaneous	5-88
arp	Displays or modifies the contents of the NetROM Address Resolution Table.	Network interface	5-14
batch	Downloads and executes a batch file containing NetROM commands.	Miscellaneous	5-89
di	Displays various "generic" NetROM state variables, statistics, and target statistics information.	Display	5-55
fill	Fills emulation memory with a known pattern.	Target interface	5-22
help	Accesses NetROM on-line help facility.	Miscellaneous	5-90
history	Displays the contents of the history buffer for the current NetROM session.	Miscellaneous	5-91
ifconfig	Displays or configures a network interface.	Network interface	5-15
kill	Sends a signal from one process running on NetROM to any other process.	Process control	5-33
ledmap	Maps NetROM's target status signals to LEDs on the back panel.	Miscellaneous	5-92

**Table 5-4** NetROM command summary (Continued)

<b>Command</b>	<b>Description</b>	<b>Type</b>	<b>Page</b>
load	Loads NetROM environment variables and IP address from non-volatile storage.	Miscellaneous	5-93
loadmodule	Loads the RAM-based optional software.	Miscellaneous	5-94
logout	Terminates a login session.	Miscellaneous	5-95
netstat	Displays network statistics.	Network interface	5-17
newimage	Downloads a file into emulation memory.	Target interface	5-23
ping	Determines whether remote hosts are up and accessible.	Network interface	5-18
printenv	Displays the current values of NetROM's environment variables.	Environment variable	5-101
ps	Displays the current status of processes running on NetROM.	Process control	5-34
reset	Resets NetROM hardware and software.	Miscellaneous	5-96
romset	Manipulates large ROM address spaces or word sizes greater than 32 bits.	ROM set	5-78
route	Manipulates information in NetROM's IP routing table.	Network interface	5-19
save	Saves NetROM environment variables and IP address to non-volatile storage.	Miscellaneous	5-97
serialcons	Creates a "console" on a non-target system using NetROM's target serial port.	Target interface	5-27
set	Sets or modifies various NetROM state variables.	Set	5-35
setenv	Modifies the value of environment variables.	Environment variable	5-101

Table 5-4 NetROM command summary (Continued)

Command	Description	Type	Page
slip	Attaches or detaches a serial line to the serial line IP interface.	Network interface	5-20
stty	Displays or modifies characteristics of NetROM terminal sessions.	Miscellaneous	5-98
tgtcons	Establishes a console session with the target system.	Target interface	5-29
tgtrset	Reinitializes communication channels and dualport pointers, and can reset target processor if an external reset line is connected.	Target interface	5-31

## Understanding the command descriptions

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 should 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 IP 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. 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.

**Table 5-5** Network interface commands.

Argument	Description	Page
arp	Displays and/or modifies the contents of NetROM's Address Resolution Table.	5-14
ifconfig	Displays or modifies the address, netmask, broadcast address, or operating state of one of NetROM's interfaces.	5-15
netstat	Displays network statistics and routing information.	5-17
ping	Sends ICMP ECHO_REQUEST packets to network hosts.	5-18
route	Manipulates information in NetROM's routing table.	5-19
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.	5-20

## arp

Displays or modifies the contents of NetROM's Address Resolution Table.

### Syntax

```
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.

### Syntax

```
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 serial 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.

### Syntax

```
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.

## Syntax

```
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.

### Syntax

```
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.

### Syntax

*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.

**Table 5-6** Target interface commands.

<b>Argument</b>	<b>Description</b>	<b>Page</b>
<code>fill</code>	Allows initialization of emulation memory to arbitrary values.	5-22
<code>newimage</code>	Downloads a file into emulation memory.	5-23
<code>serialcons</code>	Creates a "console" to a non-target system in environments wherein neither the debug path nor the console path use NetROM's target serial port.	5-27
<code>tgtcons</code>	Establishes a console session with the target system.	5-29
<code>tgtrset</code>	Reinitializes communication channels and dualport pointers, and can reset target processor if an external reset line is connected.	5-31

## fill

Allows initialization of emulation memory to arbitrary values.

### Syntax

```
fill value [ romgroup | dpmem ]
```

### Description

This command fills the emulation space of one of NetROM's ROM groups with a known value. This value is specified as an 8-bit hexadecimal number given by the **value** parameter. The optional **romgroup** parameter indicates which ROM group's emulation memory should be filled. If omitted, the default romgroup is assumed. If the optional *dpmem* keyword is used instead of a ROM 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.

### Syntax

```
newimage [filename] [ type={binary|srecord|intelhex} ]  
[ base=baseaddr ] [ offset=offset ] [ group=romgroup ]  
[ 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 ROM 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 ROM 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 ROM 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 ROM group's base. For example, if a target ROM group's base address is 0 and records in a file are addressed beginning at 0, but the file is *really* located at 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 ROM 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 “romgroup”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 “romgroup” variable determines which of the four possible ROM groups 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 ROM group and server address may be overridden using the **romgroup** 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 you want to load three files into the default ROM group's emulation memory. The files each contain 0x100 bytes of data and need to be located in the target's memory map starting at the beginning of the emulation memory. The NetROM default group's starting address in the target is 0x40000, specified by the "groupaddr" environment variable. The files have these characteristics:

- The file *file1.srec* is in S-Record format. It contains data and address information on where to store the data in memory. In this file, the first byte of data is to be stored at address 0x40000.
- The file *file2.bin* is a binary file. It contains only data and does not contain any address information.
- The file *file3.hex* is an Intel hex file. It contains data and information on where to store the data in memory. In this file, the first byte of data is to be stored at address 0x0.

The following commands are used to load these files into emulation memory:

```
NetROM> newimage file1.srec fillpattern=0xff
NetROM> newimage file2.bin offset=0x100 type=binary
NetROM> newimage file3.hex base=0x0 offset=0x200 type=intelhex
```

The first command fills NetROM emulation memory with 0xff before loading the S-record file, *file1.srec*. Since the first data address specified in the file is the same as the first address of emulation memory as defined by the "groupaddr" environment variable, the data is loaded there.

The second command loads the data in the binary file, *file2.bin*, to the start of emulation memory, offset by 0x100 bytes, as specified by the *offset* argument.

The third command loads the Intel hex file, *file3.hex*. The first byte of emulation memory is set to be at address 0x0 by the argument, *base=0x0*. This overrides the "groupaddr"

environment variable. The first byte of data, which should be stored at target address 0x0, will actually go to the beginning of NetROM emulation memory, plus any offset specified by the *offset* argument. This command will result in the first byte of file3.hex being stored at address 0x40200.

When a *newimage* command loads a file into an area of emulation memory that already contains data, the old data will be replaced with the new data. For example, if the binary file, file2.bin, were 0x200 bytes in length, then the last 0x100 bytes would have been overwritten by data loaded from file3.hex. If the *fillpattern* argument used with the first command had been used in all three commands, then after the third command, the only data in NetROM's emulation memory would be data from file3.hex. The rest of emulation space would be filled with the value 0xff.

Note



---

You 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 ROM group if using the *set rgconfig* command.

---

### See also

"verify" environment variable, page 5-140

"groupaddr" environment variable, page 5-122

"fillpattern" environment variable, page 5-121

*tgtrset* command, page 5-31

*di rgconfig* command, page 5-70

*set emulate off* command, page 5-40

*set emulate on* command, page 5-40

## serialcons

Creates a “console” to a non-target system in environments wherein neither the debug path nor the console path use NetROM’s target serial port.

### Syntax

*serialcons*

### Description

The *serialcons* command allows you 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 a 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 you might use the “dualport” console and debug paths. This would cause the *tgtcons* command, and any remote debuggers being run, to use the dualport mailbox protocol 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 you 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 5-29

“debugpath” environment variable, page 5-115



*"consolepath"* environment variable, page 5-113  
*stty* command, page 5-98

## **tgtcons**

Establishes a console session with the target system.

### **Syntax**

*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 the emulation RAM mailbox protocol. This protocol is 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 "eof" character, above.

### **See also**

*serialcons* command, page 5-27

"debugpath" environment variable, page 5-115

"consolepath" environment variable, page 5-113

*stty* command, page 5-98

## tgtrreset

Resets the target processor.

### Syntax

*tgtrreset*

### Description

When the *tgtrreset* 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 mailbox without interference from the target.
3. NetROM resets emulation memory mailbox 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 *tgtrreset* command must be invoked after changing the environment “consolepath” or “debugpath” variables. This synchronizes the change of communication protocols between the host, 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 5-29

“consolepath” environment variable, page 5-113

“debugpath” environment variable, page 5-115

*set emulate off* command, page 5-40

*set emulate on* command, page 5-40

---

## 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.

**Table 5-7** Process control commands.

<b>Argument</b>	<b>Description</b>	<b>Page</b>
kill	Sends a signal to any process running on NetROM.	5-33
ps	Displays the current status of processes running on NetROM.	5-34

## kill

Sends a signal to any process running on NetROM.

### Syntax

```
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 5-1 on page 5-3.

The **pid** 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.

### Syntax

```
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 5-33. 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 5-8 summarizes the *set* command.

**Table 5-8** Arguments to the *set* command

Argument	Description	Page
?	Displays arguments to the set command.	5-90
chanecho	Enables or disables echoing channel data passed to the target system.	5-37
consecho	Enables or disables echoing console data passed to the target system.	5-38
debugecho	Enables or disables echoing debug data passed to the target system.	5-39
emulate	Enables or disables target system access to emulation memory.	5-40



**Table 5-8** Arguments to the *set* command (Continued)

<b>Argument</b>	<b>Description</b>	<b>Page</b>
help	Displays arguments to the <i>set</i> command.	5-90
loadecho	Enables or disables debug information on downloads.	5-41
podmem	Sets values in emulation memory.	5-42
prompt	Sets the session prompt.	5-43
rgconfig	Configures a ROM group.	5-44
rgname	Assigns a name to a ROM group.	5-47
romupgrade	Installs a new version of the NetROM program.	5-48
tgtctl	Turns on or off command signals to the target system.	5-51
udpsrcmode	Sets the state of the UDP source address variable.	5-53
username	Enables an advisory lock on the NetROM unit.	5-54

## set chanecho

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

### Syntax

```
set chanecho [1 | 2 | 3] { on | off }
```

### Description

Channel echoing is a debug tool for cases in which the host system has trouble with a channel path connection to the target. When channel echoing is on for a channel (1, 2, or 3), channel data received from the target is echoed to the NetROM console session that issued the *set chanecho* command before it is forwarded to the host system, and channel data received from the host system is echoed to the NetROM console before it is forwarded to the target.

If the environment variable “chanpath #=*serial*” is set, data is displayed only on the NetROM console serial port.

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

### See also

*di debugecho* command, page 5-39

## set consecho

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

### Syntax

```
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 serial 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 5-58

## set debugecho

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

### Syntax

```
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 NetROM 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 session 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 5-59

## set emulate

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

### Syntax

```
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 ROM 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 5-62

## set loadecho

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

### Syntax

```
set loadecho { on | off }
```

### Description

Download echoing is a debug tool for cases in which the host system has trouble with its download path connection to the target. When download echoing is on, download data received from the target is echoed to the NetROM console session that issued the *set loadecho* command before it is forwarded to the host system, and download data received from the host system is echoed to the NetROM console before it is forwarded to the target.

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

### See also

*di loadecho* command, page 5-65

## set podmem

Sets values in emulation memory.

### Syntax

*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 value being written to memory. NetROM uses the **address** parameter to determine which ROM group will be affected by the write operation.

The size and endian orientation is controlled by the binary display/set environment settings. To change these settings use "setenv binenv".

When "binenv" is set to 16, 32, or 64, the **tgtaddr** must start on a word boundary. If it does not, an error message displays.

### See also

*di podmem* command, page 5-69

"binenv" environment variable, page 5-108

## set prompt

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

### Syntax

```
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 rgconfig

Completely configures a ROM group for emulation.

### Syntax

```
set rgconfig groupnum romtype tgtaddr podorder  
( readonly | readwrite)
```

### Description

The *set rgconfig* command is used to completely define a ROM group 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 ROM groups using environment variables than with the *set rgconfig* 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 rgconfig* command.

The **groupnum** parameter indicates which ROM group is being configured. If the “special” ROM group named by the “romgroup” environment variable is being configured, NetROM will change environment variables to agree with the command-line specification for this command. Refer to “Emulation memory” on page 2-11.

The **romtype** parameter is the name of the ROM type being emulated. Valid ROM types are given by Table 5-15 on page 5-136.

The **tgtaddr** parameter is the 32-bit base address of the ROM 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 5-127. If the specification of pods in the `podorder` parameter conflicts with pods in use by the ROM group named by the “`romgroup`” 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 ROM group, using the external write line.

### Examples

```
set rgconfig 0 27c020 bfc00000 0:1:3:2 readonly
```

Configures ROM group 0 to emulate 27c020 ROMs. The ROM 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 rgconfig 0 27c010 0 1:0-3:2 readwrite
```

Configures ROM group 0 to emulate 27c010 ROMs. The ROM 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 rgconfig 0 27c010 0 1-0-2-3 readonly
```

Configures ROM group 0 to emulate 27c010 ROMs. The ROM 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 ROM 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

*"romgroup"* environment variable, page 5-135  
*"romtype"* environment variable, page 5-136  
*"groupaddr"* environment variable, page 5-122  
*"romcount"* environment variable, page 5-134  
*"wordsize"* environment variable, page 5-142  
*"podorder"* environment variable, page 5-127  
*di podmem* command, page 5-69  
*di rgconfig* command, page 5-70  
*set rgname* command, page 5-47  
*setenv* command, page 5-104  
*printenv* command, page 5-105

## set rname

Assigns a name to a ROM group.

### Syntax

```
set rname namestring [ romgroup ]
```

### Description

The *set rname* command assigns a name to a ROM group. This ROM group must have either been configured using environment variables or with the *set rgconfig* command. ROM group names are optional and are essentially mnemonics to help you remember what the ROM group is emulating, if more than one ROM group is in use.

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

The *set rname* command will not work on ROM groups which have not yet been configured.

### See also

*set rgconfig* command, page 5-44

*di rgconfig* command, page 5-70

## set romupgrade

Initiates the download of a new NetROM operating system image.

### Syntax

```
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, *rom.bin* and *netrom.bin*; *rom.bin* is the new system image and *netrom.bin* is a RAM-based download program. 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 *rom.bin* into FLASH ROM memory in your NetROM unit automatically. When the download is complete, NetROM resets itself.

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. 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. A TFTP client has been created, run for a while, and exited. You can see this process using the *ps* command. NetROM is verifying that *rom.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, *rom.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 no 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

Controls NetROM's control signal outputs.

### Syntax

```
set tgtctl signum {on | off} [millisec_interval]  
set tgtctl signum {on | off} [toggle]  
set tgtctl signum {on | off} [rx | rx0 | rx1 | rx2 |  
rx3] [ack]
```

### Description

This command asserts or de-asserts one of the control signals on NetROM's front panel command status connector. 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.

When *millisec\_interval* is specified, 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 varies depending on NetROM's clock divider. Use *? set tgtctl* to display the value of the interval. Accuracy of the timing is dependent on NetROM's load and may vary. Don't use the timer for measurements that require a high degree of accuracy. The timer will run until the signal is turned on or off without a new interval.

When *toggle* is specified, 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.

If *ack* is specified, NetROM won't trigger a subsequent interrupt if the previous one has not been acked. When *rx* is specified, the signal will be asserted when data is passed to the target using the dualport emulation memory protocol on any channel. This signal can be attached to the target system, causing an interrupt to the target when data is ready to be read. To specify a specific dualport channel, use:



<b>rx#</b>	<b>Dualport channel</b>
<i>rx0</i>	Console path/Debug path
<i>rx1</i>	Channel 1
<i>rx2</i>	Channel 2
<i>rx3</i>	Channel 3

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

See Chapter 5 in the *NetROM Hardware Interface Reference* manual for the names of NetROM's target control signals (*signum*) relative to their pin numbers on the command status connector.

## set udpsrcmode

Enables connectionless debug sessions.

### Syntax

```
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 path 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.

## **set username**

Sets an advisory login lock on the NetROM unit.

### **Syntax**

*set* **username**

### **Description**

The *set username* command enables an advisory login lock on the NetROM unit.

## 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 5-9 summarizes the *di* command.

**Table 5-9** *di* command arguments

Argument	State or statistics displayed	Page
?	List of <i>di</i> arguments and what they display.	5-90
chanecho	Channel echo state, on or off.	5-57
consecho	Console echo state, on or off.	5-58
debugecho	Debug echo state, on or off.	5-59
loadecho	Load echo state.	
dpmem	Contents of dualport RAM.	5-60
dpstats	Statistics for dualport protocol.	5-61
emulate	Target access to emulation memory.	5-62
help	List of <i>di</i> arguments and what they display.	5-90
lanceha	NetROM's Ethernet address.	5-63
ledmap	Mapping between NetROM status signals and back panel LEDs.	5-64
lstats	Ethernet statistics.	5-66
memstats	Memory use statistics.	5-67
modules	Names of optional RAM modules loaded.	5-68

**Table 5-9** *di* command arguments (Continued)

<b>Argument</b>	<b>State or statistics displayed</b>	<b>Page</b>
<code>podmem</code>	Contents of emulation memory.	5-69
<code>rgconfig</code>	ROM group configurations.	5-70
<code>tgtctl</code>	State of NetROM's command signals.	5-71
<code>tgtstatus</code>	State of NetROM's status signals.	5-72
<code>uart</code>	Statistics for NetROM's serial ports.	5-73
<code>udpsrcmode</code>	Current state of the UDP debug source address variable.	5-74
<code>uptime</code>	Time since the last system reset.	5-75
<code>username</code>	User name used for advisory login locks.	5-76
<code>version</code>	Software version number for NetROMs operating system.	5-77

## di chanecho

Displays whether channel echoing is turned on or off.

### Syntax

```
di chanecho {1 | 2 | 3}
```

### Description

*di chanecho* prints to the screen the current state of NetROM's channel echo variable for the specified channel (1, 2, or 3). To change the variable, use the *set chanecho* command.

### See also

*set chanecho* command, page 5-37

## **di consecho**

Displays whether console echoing is turned on or off.

### **Syntax**

*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 5-38

## di debugecho

Displays whether debug echoing is turned on or off.

### Syntax

*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 5-39



## di dpmem

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

### Syntax

```
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 ROM group of which pod 0 is a part.

### See also

*di podmem* command, page 5-69

“Dualport emulation memory” on page F-4

“Dualport protocol” on page F-10

## di dpstats

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

### Syntax

*di dpstats*

### Description

This command displays statistics about the dualport protocol on channels 0..3 used to transmit data between NetROM and the target system.

Statistics for the protocol include whether the channel is enabled, 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 *tgtrereset* command, page 5-31.

## **di emulate**

Displays whether ROM emulation is turned on.

### **Syntax**

*di emulate*

### **Description**

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

### **See also**

*set emulate* command, page 5-40

## di lanceha

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

### Syntax

*di lanceha*

### Description

The *di lanceha* command displays 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**

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

### **Syntax**

*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 5-92

## di loadecho

Displays whether load echoing is turned on or off.

### Syntax

*di loadecho*

### Description

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

### See also

*set loadecho* command, page 5-41

## **di lstats**

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

### **Syntax**

*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.

### Syntax

*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.

### **Syntax**

*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 podmem

Displays the contents of emulation memory.

### Syntax

*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 ROM group should be displayed. The size and endian orientation is controlled by the binary display/set environment settings. To change these settings use "setenv binenv".

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, an error message displays.

When "binenv" is set to 16, 32, or 64, the **tgtaddr** is set to the nearest word boundary, if the **tgtaddr** given is not on a word boundary.

### See also

*set podmem* command, page 5-42

*set rgconfig* command, page 5-44

*set emulate* command, page 5-40

"binenv" environment variable, page 5-108

## di rgconfig

Displays the current ROM group configurations.

### Syntax

```
di rgconfig [ romgroup ]
```

### Description

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

### See also

*set rgconfig* command, page 5-44

*romgroup* command, page 5-135

## di tgtctl

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

### Syntax

```
di tgtctl
```

### Description

This command displays the current state of target control signals on the command status connector 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. 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 the dualport protocol.

See Chapter 5 in the *NetROM Hardware Interface Reference* manual for the names of NetROM's target control signals relative to their pin numbers on the command status connector.

### See also

*set tgtctl* command, page 5-51

## **di tgtstatus**

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

### **Syntax**

*di tgtstatus*

### **Description**

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

## di uart

Displays statistics for NetROM's serial port UARTs.

### Syntax

```
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**

Displays the state of the UDP source address mode variable.

### **Syntax**

*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 5-53

## di uptime

Displays the amount of time since the last system reset.

### Syntax

*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 has installed an advisory login lock on the NetROM unit.

### **Syntax**

*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 NetROM's version numbers.

### **Syntax**

*di version*

### **Description**

The *di version* command displays the NetROM model, hardware ID, software version number, date and time of NetROM's operating system.

---

## ROM set commands

For target systems which require large ROM address spaces or word sizes greater than 32 bits, a 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.

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;
- 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 5-10** *romset* command arguments

<b>Argument</b>	<b>Description</b>	<b>Page</b>
?	Displays arguments to <i>romset</i> command.	
clear	Clears current <i>romset</i> definitions.	5-80
connect	Connects to slave units and enters <i>romset</i> mode.	5-81
define	Defines the <i>romset</i> pod order.	5-82
disconnect	Disconnects from slave units and returns to normal mode.	5-83
help	Displays arguments to <i>romset</i> command.	
show	Displays the current <i>romset</i> configuration.	5-84
slaveaddr	Sets the addresses of slave units.	5-85
reset	Resets all slave units.	5-86

## romset clear

Erases ROM set definitions.

### Syntax

```
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 5-82

*romset slaveaddr* command, page 5-85



## romset connect

Causes NetROM to create TCP connections with slave units.

### Syntax

*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 5-85

*romset disconnect* command, page 5-83

## romset define

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

### Syntax

```
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.

### Syntax

*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.

### Syntax

*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

Assigns the network addresses of ROM set slave units.

### Syntax

```
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 5-82

## romset reset

Causes all slave units to reset themselves.

### Syntax

```
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.

**Table 5-11** Miscellaneous commands

<b>Command</b>	<b>Description</b>	<b>Page</b>
<code>?</code>	List of <i>di</i> arguments and what they display.	5-90
<code>alias</code>	Creates command "nicknames".	5-88
<code>batch</code>	Downloads and executes batch files.	5-89
<code>help</code>	List of <i>di</i> arguments and what they display.	5-90
<code>history</code>	Displays the contents of the command history buffer.	5-91
<code>load</code>	Load NetROM environment variables and IP address from non-volatile storage.	5-93
<code>ledmap</code>	Maps NetROM's status signals to LEDs.	5-92
<code>loadmodule</code>	Loads NetROM's optional RAM module.	5-94
<code>logout</code>	Terminates login sessions.	5-95
<code>reset</code>	Resets NetROM's hardware and software.	5-96
<code>save</code>	Saves NetROM environment variables and IP address to non-volatile storage.	5-97
<code>stty</code>	Displays or modifies characteristics of NetROM terminal sessions.	5-98

## alias

Creates and deletes command “nicknames.”

### Syntax

```
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.

### Syntax

```
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 5-6 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.

### Syntax

*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.

### Syntax

*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 5-5 for details on history substitution.



# ledmap

Maps NetROM's status signals to LEDs on the back panel.

## Syntax

```
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 front panel (see Figure 2-5). 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 5-64

## load

Lloads NetROM environment variables and IP address from non-volatile storage.

### Syntax

*load*

### Description

After you *save* the environment variables to non-volatile memory, you may later recall them using the *load* command. To automatically load your values when NetROM is power cycled, use the *setenv bootflags* command.

### See also

"bootflags" environment variable, page 5-109  
*save* command, page 5-97

## loadmodule

Loads NetROM's optional RAM modules.

### Syntax

```
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 5-68

## logout

Terminates login sessions.

### Syntax

*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.

### **Syntax**

*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.

## save

Saves NetROM environment variables and IP address to non-volatile storage.

### Syntax

*save*

### Description

After you *save* the environment variables to non-volatile memory, you may later recall them using the *load* command. To automatically load your values when NetROM is power cycled, see the *setenv bootflags* command.

### Note



---

The "loadpath" and "batchpath" environment variables are limited to 60 bytes in length. The "loadfile" and "batchfile" environment variables are limited to 24 bytes in length.

---

### See also

"bootflags" environment variable, page 5-109  
*load* command, page 5-93

## stty

Displays or modifies characteristics of NetROM terminal sessions.

### Syntax

```
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 5-4 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 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 consolebaud=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 5-29

## 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 ROM groups for emulation, and setting default values for downloading new emulation images.

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.

Table 5-12 summarizes the environment variables.

**Table 5-12** Environmental variables

Variable	Description	Page
"batchfile"	Specifies default batch file to process.	5-106
"batchpath"	Sets path on the TFTP file server NetROM uses to search for batch files and RAM module locations.	5-107
"binenv"	Controls how emulation memory is written and displayed.	5-108
"bootflags"	Controls NetROM's boot-time behavior	5-109
"chanpath"	Sets channel communication path between NetROM and the target system.	5-110
"chanport"	Sets TCP/UDP port number on which NetROM accepts communications on channel path.	5-112
"consolepath"	Sets console communication path between NetROM and the target system.	5-113

**Table 5-12** Environmental variables (Continued)

<b>Variable</b>	<b>Description</b>	<b>Page</b>
“debugpath”	Sets debug communication path between NetROM and the target system.	5-115
“debugport”	Sets TCP/UDP port number on which NetROM accepts communications on debug path.	5-117
“dprbase”	Sets base address in emulation pod 0 to map dualport RAM.	5-118
“filetype”	Sets expected download file format. Supports binary, S-record, and Intel hex.	5-120
“fillpattern”	Sets byte pattern to fill emulation memory.	5-121
“groupaddr”	Sets default ROM group’s start address.	5-122
“groupwrite”	Enables or disables NetROM’s external write signal.	5-123
“host”	Sets IP address of the TFTP server used for image and batch downloads, <i>romupgrade</i> , and <i>loadmodule</i> .	5-124
“loadfile”	Sets default file to download into the default ROM group.	5-125
“loadpath”	Sets default path for downloading “loadfile” and <i>romupgrade</i> .	5-126
“podorder”	Sets pod-to-byte mapping of emulation pods in the default ROM group.	5-127
“romcount”	Sets number of bytes in emulation as part of the default ROM group.	5-134
“romgroup”	Sets default ROM group.	5-135
“romtype”	Sets ROM type being emulated by the default ROM group.	5-136
“tgtip” (optional)	Sets target machine’s IP address when Virtual Ethernet is on.	5-139

**Table 5-12** Environmental variables (Continued)

<b>Variable</b>	<b>Description</b>	<b>Page</b>
“verify”	Specifies whether downloads are verified. Takes as values: <i>on</i> or <i>off</i> .	5-140
“vether” (optional)	Sets Virtual Ethernet on or off.	5-141
“wordsize”	Sets size in bits of the ROM word being emulated by default ROM group.	5-142
“writemode”	Sets write mode that configures emulation memory to emulate FLASH ROM or static RAM.	5-143

## setenv

Modifies the value of environment variables.

### Syntax

```
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 5-12 summarizes the names and characteristics of NetROM's environment variables. The format of the **value** parameter depends on the variable being set. Consult the variable descriptions for more information on how to specify the value.

## **printenv**

Displays the current values of NetROM's environment variables.

### **Syntax**

*printenv*

### **Description**

The *printenv* command displays the current settings for all environment variables. The variables are summarized in Table 5-12, and discussed in detail in the documentation that follows.

## batchfile

Specifies default batch file to process.

### Syntax

```
setenv batchfile filename
```

### Description

The “batchfile” environment variable is the name of the default batch file to process. This should be a simple file name, not a directory path. This file name is concatenated with the “batchpath” environment variable to determine the root-specific path of the default batch file.

The “batchfile” default can be overridden on the command line, when invoking *batch*, if desired. Note that setting “batchpath” to “/” effectively clears it on secure servers, and setting it to “/tftpboot” clears it on non-secure servers.

### See also

*batch* command, page 5-89

“batchpath” environment variable, page 5-107

## batchpath

Sets the path on the TFTP server for the batch file.

### Syntax

```
setenv batchpath path
```

### Description

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.

The “batchpath” environment variable is specifically designed to facilitate the *batch* command. You may have certain commands that you 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 you can cause NetROM to download these files and execute the commands in them one at a time.

### See also

*batch* command, page 5-89

“batchfile” environment variable, page 5-106



## binenv

Specifies how emulation memory is read and displayed.

### Syntax

```
setenv binenv { 8 | 16 | 32 | 64 } { big | little }
```

### Description

The "binenv" environment variable controls how the *set podmem* and *di podmem* commands write and display NetROM emulation memory.

Select *8*, *16*, *32*, or *64* to specify the word length, in bits, to use when writing or displaying data. The default is 8 bit, no endian.

Select *big* or *little* to specify which endian mode to use when writing or displaying emulation memory at the NetROM prompt.

### See also

*set podmem* command, page 5-42

*di podmem* command, page 5-69

## bootflags

Controls NetROM's boot-time behavior.

### Syntax

```
setenv bootflags [loadenv] [rarp] [bootp] [storage]
[autobat|userbat] [hostip]
```

### Description

This command sets the flags that determine how NetROM configures its environment when power-cycled. To enable a feature, include the appropriate flag on the command line. Flags may be specified using one or more letters: *loadenv*, *load*, and *l* are equivalent.

**Table 5-13** Bootflags and their effects

Option	Action
loadenv	Auto-load the previously <i>saved</i> environment when NetROM is power-cycled
rarp	RARP for NetROM's IP address
bootp	Use BOOTP to find NetROM's IP address
storage	Use the previously <i>saved</i> NetROM IP address
autobat	Use NetROM's IP address in hexadecimal as the batch file name. Example: IP address is 128.1.2.3, batchfile will be 80010203.
userbat	Use previously <i>saved</i> batchfile name
hostip	Use previously <i>saved</i> host IP address

NetROM's factory default for "bootflags" is:

```
setenv bootflags rarp bootp autobat
```

To load all values from storage, use:

```
setenv bootflags l s h
```

## chanpath

Sets a channel communication path between NetROM and the target system.

### Syntax

```
setenv chanpath {1 | 2 | 3} {serial | dualport}
```

### Description

The “chanpath” environment variable is used to configure channel path communications between NetROM and the target system. Channel 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.

Select *1*, *2*, or *3* to specify which channel you are setting up.

Select *serial* or *dualport* to specify the communication protocol for the selected channel.

The *serial* value selects NetROM’s target serial port interface for the channel path communication between NetROM and the target. Parameters for this port are configured using the *stty* command (see “stty” on page 5-98).

The *dualport* value selects the emulation memory mailbox protocol for channel # path communication between NetROM and the target. The mailbox protocol is described in detail in Chapter 8.

### Note



---

Changing the “chanpath” variable will not take immediate effect; the target must be reset with the *tgreset* command before the NetROM-target communication path will change. This prevents corruption of any current active channel sessions.

---

The "channelpath" variable is independent of the "consolepath" and the "debugpath" variables. Each channel path has its own dualport channel. In situations where more than one channel is communicating serially, the host will receive all communications and must distinguish between them.

## chanport

Sets TCP/UDP port number on which NetROM accepts data from host.

### Syntax

```
setenv chanport {1 | 2 | 3} portnum
```

### Description

The “chanport” environment variable is used to set the TCP/UDP port number on which NetROM listens for data from the host. The **portnum** parameter should be in decimal.

channel #	portnum default
1	1240
2	1241
3	1242

### Note



If a session is underway, changing the “chanport” variable will not take immediate effect; the target must be reset with the *tgtrreset* command before the NetROM-target communication port will change. This prevents corruption of any current active channel sessions.

### See also

*set udpsrcmode* command, page 5-53

## consolepath

Sets the console communication path between NetROM and the target system.

### Syntax

```
setenv consolepath {serial | dualport}
```

### Description

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.

Select *serial* or *dualport* to specify the communication protocol for the console path.

The *serial* value selects NetROM’s target serial port interface for the console path communication between NetROM and the target. Parameters for this port are configured using the *stty* command (see “stty” on page 5-98).

The *dualport* value selects the emulation memory mailbox protocol for the console path communication between NetROM and the target. The mailbox protocol is described in detail in Chapter 8.

### Note



---

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

---

The “consolepath” variable and the “debugpath” variable are independent, because one may use the serial port communications path and the other the mailbox protocol, or they may both use the mailbox protocol or the serial port. 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 5-115.

## debugpath

Sets the debug communication path between NetROM and the target system.

### Syntax

```
setenv debugpath {serial | dualport}
```

### Description

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.

Select *serial* or *dualport* to specify the communication protocol for the debug path.

The *serial* value selects NetROM’s target serial port interface for the debug path communication between NetROM and the target. Parameters for this port are configured using the *stty* command (see “stty” on page 5-98).

The *dualport* value selects the emulation memory mailbox protocol for the debug path communication between NetROM and the target. The mailbox protocol is described in detail in Chapter 8.

### Note



---

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

---

The “consolepath” variable and the “debugpath” variable are independent, because one may use the serial port communications path and the other the mailbox protocol, or they may both use the mailbox protocol or the serial port. 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 “consolepath” on page 5-113.

## debugport

Sets TCP/UDP port number on which NetROM accepts data from host-based debuggers.

### Syntax

```
setenv debugport portnum
```

### Description

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



---

If a session is underway, changing the “debugport” variable will not take immediate effect; the target must be reset with the *tgreset* command before the NetROM-target communication port will change. This prevents corruption of any current active debug sessions.

---

### See also

*set udpsrcmode* command, page 5-53

## dprbase

Sets base address in emulation pod 0 to map dualport RAM.

### Syntax

```
setenv dprbase offset
```

### Description

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 ROM group containing pod 0. That is, it should be considered the byte offset from the start of the ROM which pod 0 is emulating. “dprbase” must start on an 8K boundary.

The dualport requires 8K of memory for passing messages. By default, the dualport memory is mapped to the last 8192 bytes of memory emulated by pod 0.

For example, for a 4 MB NetROM, if pod 0 is emulating an at27c080 ROM, which has 1 M, the default “dprbase” is 0xfe000, since this is 8192 (or 0x2000) bytes below the end of ROM (0xffff).

For a 1 MB NetROM, if pod 0 is emulating a 27c020 ROM, which has 256K, the default “dprbase” is 0x3e000, since this is 8192 (or 0x2000) bytes below the end of the ROM (0x3fff).

The “dprbase” variable allows you 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. 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.

**Note**

---

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 *tgtrreset* command.

---

## filetype

Sets expected download file format.

### Syntax

```
setenv filetype format
```

### Description

The “filetype” environment variable tells NetROM what file format to expect when downloading the default ROM group (which is named by the “romgroup” environment variable). Supported file types and their associated settings are given in Table 5-14. The “filetype” default can be overridden on the command line when invoking *newimage*.

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 5-14** Supported NetROM file formats

<b>format</b>	<b>Meaning</b>
binary	Binary file
srecord	Motorola S-record file
intelhex	Intel hex record file



## fillpattern

Sets byte pattern to fill emulation memory.

### Syntax

```
setenv fillpattern pattern
```

### Description

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 **pattern** are either the word *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 “newimage” on page 5-23 for details.

## groupaddr

Sets default ROM group's start address.

### Syntax

```
setenv groupaddr address
```

### Description

The “groupaddr” environment variable gives NetROM the *target's* idea of the start address of the default ROM group (which is named by the “romgroup” 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

Enables or disables NetROM's write signal.

### Syntax

```
setenv groupwrite {readonly | readwrite}
```

### Description

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. Appropriate values for the “groupwrite” variable are *readonly* and *readwrite*.

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.



## host

Sets IP address of the TFTP server used for image and batch downloads.

### Syntax

```
setenv host ip_address
```

### Description

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 can be overridden on the command line, for example when invoking the *batch* and *newimage* commands.

## loadfile

Sets default file to download into the default ROM group.

### Syntax

```
setenv loadfile filename
```

### Description

The “loadfile” environment variable is the name of the default file to download into the default ROM group (which is named by the “romgroup” 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

Sets default path for downloading "loadfile".

### Syntax

```
setenv loadpath path
```

### Description

The "loadpath" environment variable is the default directory path NetROM will use when downloading image files into the default ROM group (which is named by the "romgroup" 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



---

For 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.

---

## podorder

Sets pod-to-byte mapping of emulation pods in the default ROM group.

### Syntax

```
setenv podorder pod#(-|:)pod# ...
```

### Description

The “podorder” environment variable (see Figure 5-1) maps emulation pods within the default ROM group (which is named by the “romgroup” environment variable) to ROM sockets being emulated by that ROM group.

For example, if ROM group 0 uses pods 0 and 1 and emulates a 16-bit-word target ROM space for two ROMs, 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 0-3 as follows:

- separated by colons (':') indicates “parallel” usage (pods emulate multiple ROMs used for single words)
- separated by dashes ('-') indicates “serial” usage (pods emulate multiple ROMs used for consecutive words)

For passive cables, each emulation pod supports a single ROM and must be mapped separately.

For active cables, the emulation pods are used in pairs: 0 and 1 are used together and 2 and 3 are used together. Specify 0 to use pods 0 and 1; specify 2 to use pods 2 and 3.

### Examples - passive cables

- 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 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; “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.

#### Example - active cables

- The notation “0:2” indicates that pods 0/1 and 2/3 work together. If you are using 8-bit ROMs, this would emulate a 16-bit word, in which pods 0/1 emulate byte 0 and pods 2/3 emulate byte 1.
- The notation “0-2” indicates that pods 0/1 and 2/3 work together. Pods 0/1 emulate the lower-addressed words and pods 2/3 emulate the higher-addressed words. This notation indicates an emulated space where words are half as wide as the first notation, but in which there are twice as many words.

#### Note



---

When using an active emulation cable for 16-bit devices, you cannot use *podorder* to swap bytes within the device, although bytes can be swapped between devices. The host utility, *rompack*, (see page 6-7) can be used to perform byte swaps within a device.

---

The “podorder” variable is related to the “romcount” and the “wordsize” variables. The “podorder” variable specifies the combined information of the “romcount” and “wordsize” variables. The “podorder” variable will override both the “romcount” and “wordsize” variables, if set. However, the “romcount” and “wordsize” variables are probably more intuitive to use. If the “podorder” variable is not set, the order of pods within the default ROM group is always the same.

Figure 5-2 through Figure 5-3 summarize the interactions of the “podorder”, “romcount”, and “wordsize” variables, and give

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.

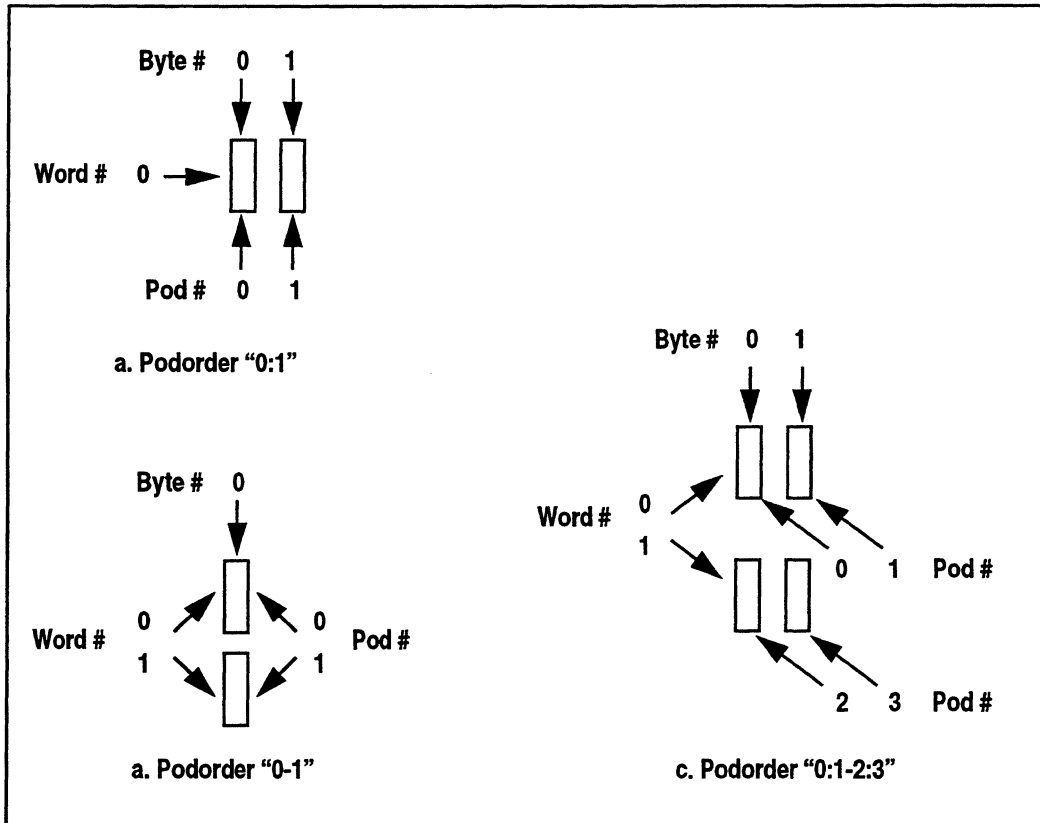


Figure 5-1 Podorder examples (passive cables)


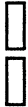

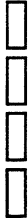






		Romcount			
		1	2	3	4
Wordsize	8	 "0"	 "0-1"	 "0-1-2"	 "0-1-2-3"
	16	No	 "0:1"	No	 "0:1-2:3"
	32	No	No	No	 "0:1:2:3"

Figure 5-2 Podorder/romcount/wordsize interactions (passive cables)


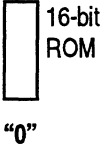
		Romcount			
		1	2	3	4
Wordsize	8	 "0"	 "0:2"	No	No
	16	No	 "0:2"	No	No
	32	No	No	No	No

**Figure 5-3** Podorder/romcount/wordsiz interactions  
(active cables - Figure 3-4A) - 8-bit ROMs



		Romcount			
		1	2	3	4
Wordsize	8	No	No	No	No
	16	□ "0"	□ □ "0-2"	No	No
	32	No	□ □ "0:2"	No	No

**Figure 5-4** Podorder/romcount/wordsize interactions  
(active cables - Figure 3-4A) 16-bit ROMs

		Romcount			
		1	2	3	4
Wordsize	8		No	No	No
	16		No	No	No
	32	No	No	No	No

**Figure 5-5** Podorder/romcount/wordsize interactions  
(active cables - Figure 3-4B)

## romcount

Sets number of ROMs being emulated in the default ROM group.

### Syntax

```
setenv romcount value
```

### Description

The “romcount” environment variable specifies the number of ROMs being emulated by the default ROM group (which is named by the “romgroup” environment variable). The “romcount” variable is related to the “podorder” and the “wordsize” variables.

The “romcount” variable is related to the “podorder” and the “wordsize” variables. The “podorder” variable specifies 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. If the “podorder” variable is not set, the order of pods within the default ROM group is always the same.

Figure 5-2 through Figure 5-3 summarize the interactions of the “podorder”, “romcount”, and “wordsize” variables, and give the default values of the “podorder” variable for each case.

## romgroup

Sets the default ROM group.

### Syntax

```
setenv romgroup value
```

### Description

The “romgroup” environment variable selects the ROM group that should be considered the default ROM group for the commands that affect ROM groups. These commands include *newimage* and *di podmem*, among others. In addition, the default ROM group is the one which is acted upon by the environment variables affecting pod group (“groupaddr”, “groupwrite”, “podorder”, “romcount”, “romgroup”, “romtype”, “wordsize”) and downloading.

## romtype

Sets the type of device being emulated by the default ROM group.

### Syntax

```
setenv romtype device
```

### Description

The “romtype” environment variable specifies the type of ROM that NetROM will emulate. Table 5-15 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 16- and 32-bit wide devices requires the use of active cables.

If your device is not listed, use the generic type that applies to your device.

**Table 5-15** ROM types and sizes

<b>ROM type</b>	<b>Size</b>	<b>ROM type variable name</b>
27c256, 28f256	32K x 8	27c256, 28f256 <b>32k_by_8</b>
27c512, 28f512	64K x 8	27c512, 28f512 <b>64k_by_8</b>
27c010, 27c100, 28f010, 28f001b	128K x 8	27c010, 27c100, 28f010, 28f001b <b>128k_by_8</b>
27c020, 28f020	256K x 8	27c020, 28f020 <b>256k_by_8</b>
27c040, 27x040, 29f040, am29f040, am29lv004b	512K x 8	27c040, 27x040, 29f040, am29f040, am29lv004b <b>512k_by_8</b>

Table 5-15 ROM types and sizes

ROM type	Size	ROM type variable name
28c400, 28f200bx, 28f400bx, 29f100b, 29f200b, 29f400ab ( <i>byte pin</i> )	512K x 8	28c400, 28f200bx, 28f400bx, 29f100b, 29f200b, 29f400ab <b>512k_by_8bp</b>
at27c080, at27lv080, m27c801, 28f008bv, 28f008sa, 28f008sc, 28f008s5, m28f841, m28v841, am29f080, 49f080, at49f080	1 M x 8	at27c080, at27lv080, m27c801, 28f008bv, 28f008sa, 28f008sc, 28f008s5, m28f841, m28v841, am29f080, 49f080, at49f080 <b>1M_by_8</b>
28f800bv_by_8, 28f800cv_by_8, am29lv800, am29lv800t ( <i>byte pin</i> )	1 M x 8	28f800bv_by_8, 28f800cv_by_8, am29lv800_by_8, am29lv800t_by_8 <b>1M_by_8bp</b>
m28f161, m28v161, 29f016, am29f016, 49f016, at49f016	2 M x 8	m28f161, m28v161, 29f016, am29f016, 49f016, at49f016 <b>2M_by_8</b>
28f032sa ( <i>byte pin</i> )	4 M x 8	28f032sa_by_8 <b>4M_by_8bp</b>
27c1024, 27c210	64K x 16	27c1024, 27c210 <b>64k_by_16</b>
27c2048, 27c220	128K x 16	27c2048, 27c220 <b>128k_by_16</b>
27c240, 27c4096	256K x 16	27c240, 27c4096 <b>256k_by_16</b>
27c400, 28f200bx, 28f400bx, 29f100b, 29f200b, 29f400ab ( <i>byte pin</i> )	256K x 16	27c400_by_16, 28f200bx_by_16, 28f400bx_by_16, 29f100b_by_16, 29f200b_by_16, 29f400ab_by_16 <b>256k_by_16bp</b>

**Table 5-15** ROM types and sizes

<b>ROM type</b>	<b>Size</b>	<b>ROM type variable name</b>
at49f8192, at27c8192	512K x16	at49f8192, at27c8192 <b>512k_by_8</b>
28f800bv, 28f800cv, am29lv800t <i>(byte pin)</i>	512K x16	28f800bv_by_16, 28f800cv_by_16, am29lv800t_by_16 <b>512k_by_8bp</b>
msm27c16227b	1 MB x 16	msm27c16227b <b>1M_by_16</b>
m27c160, 28f016sa, 28f016sv, hn625316 <i>(byte pin)</i>	1 MB x 16	m27c160_by_16, 28f016sa_by_16, 28f016sv_by_16, hn625316_by_16 <b>1M_16bp</b>
28f032sa <i>(byte pin)</i>	2 MB x 16	28f032sa_by_16 <b>2M_16bp</b>

## **tgtip (optional)**

Sets the target's IP address when Virtual Ethernet is on.

### **Syntax**

```
setenv tgtip ip_address
```

### **Description**

The “**tgtip**” environment variable specifies the target's IP address when Virtual Ethernet is turned on (see “**wordsize**” on page 5-142). 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**

Enables or disables download verification.

### **Syntax**

```
setenv verify {on |off}
```

### **Description**

The “verify” environment variable affects NetROM’s behavior as it downloads ROM 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)**

Sets Virtual Ethernet on or off.

### **Syntax**

```
setenv vether {on | off}
```

### **Description**

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.

## wordsize

Sets the size, in bits, of the ROM word being emulated by the default ROM group.

### Syntax

```
setenv wordsize {8 | 16 | 32}
```

### Description

The “wordsize” environment variable sets the width in bits of the words emulated by the default ROM group (which is named by the “romgroup” environment variable). Valid word sizes are 8, 16, and 32 bits.

The “wordsize” variable is related to the “romcount” and the “podorder” variables. The “podorder” variable specifies 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. If the “podorder” variable is not set, the order of pods within the default ROM group is always the same.

Figure 5-2 through Figure 5-5 (pages 5-130 through 5-133) summarize the interactions of the “podorder”, “romcount”, and “wordsize” variables, and give the default values of the “podorder” variable for each case.

## writemode

Sets emulation memory to emulate flash ROM or static RAM.

### Syntax

```
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.



## Chapter 6

# Utilities

Several Unix utilities are provided on your *NetROM Drivers and Utilities* diskette.

Table 6-1 summarizes the utilities alphabetically and gives their function and the page number of a complete description.

**Table 6-1** NetROM utilities summary

Utility	Description	Page
download	Downloads S-record, Intel hex, and binary images from Unix host to NetROM	6-3
ieeeparse	Creates a binary image from an IEEE record.	6-5
rompack	Modifies and/ or combines S-record, Intel hex, or binary images into a binary image for loading into NetROM. Allows byte swapping in 16, 32, and 64 bit granularity.	6-7
thost	Host-side of turboLoader application; downloads S-record, Intel hex, and binary images from Unix host to target RAM.	6-10
ttarget	Target-side of turboLoader application; downloads S-record, Intel hex, and binary images from Unix host to target RAM. ttarget must be integrated with your target code.	6-12
upload	Uploads binary images from NetROM to Unix host.	6-14

## Understanding the utility descriptions

The syntax:

```
download [-b base] [-o offset] [-f fillpattern]  
[-g podgroup] filename netromaddr
```

describes the download utility, for which the *-b* argument is optional, but which requires a base address if used. The *-o*, *-f* and *-g* arguments are also optional, and each requires a number. The **filename** and **netromaddr** are not optional, and must be supplied. **netromaddr** may be supplied either as a logical name, as in 'netrom22', or as an IP address.

When describing IP addresses, NetROM commands use standard Internet "dotted-decimal" notation. An example of such an address is "192.0.0.210".



## download

Sends an S-record, Intel hex, or binary image from the Unix host to NetROM overlay memory.

### Syntax

```
download [-b base] [-o offset] [-f fillpattern | none]
[-g podgroup] file_name netrom_address
```

### Options

<b>base</b>	base address for image; default is 0
<b>offset</b>	address offset for image; default is 0
<b>fillpattern</b>	pattern to prefill pod memory with before downloading image; default is <i>none</i> , does not alter pod memory
<b>podgroup</b>	pod group to download image to; default is 0
<b>file_name</b>	filename of image to be downloaded
<b>netrom_address</b>	network address used by NetROM

### Description

The *download* utility uses a TCP/IP socket connection between host and NetROM to download code and data images to NetROM. *download* understands S-record, Intel hex, and binary file formats.

### Compiling source code

*download.c* was compiled under SunOS 4.1.x using *gcc* version 2.6.3, using the command line:

```
gcc -o download -ansi -Wall -pedantic download.c
```

The following *gcc* flag is optional:

```
D__USE_FIXED_PROTOTYPES__
```

### Examples

Download a binary file named **rom.bin** to a NetROM with logical address **netrom22**.

```
download rom.bin netrom22
```



Download an Intel hex file named **boot.hex** to a NetROM with IP address **128.9.233.5**.

```
download boot.hex 128.9.233.5
```

Download an S-record file with a start address of **0xF0000000**, named **rom.srec**, to a NetROM with logical address **burton7**.

```
download -b 0xF0000000 rom.srec burton7
```



## ieeeparse

Creates a binary image from an IEEE record.

### Syntax

```
ieeeparse [-r romtype] [-c romcount] [-f { fillpattern
| none }] [-x] [-p] [-o outfile] file1 base1 offset1
[file2 base2 offset2 ...]
```

### Options

<b>romtype</b>	type of ROM image is to be placed in; see page 5-136 for valid values
<b>romcount</b>	number of ROMs of the same romtype that are being used in emulation
<b>fillpattern</b>	pattern to prefill pod memory with before loading images; default is <i>none</i> , which prefills memory with 00
<b>-x</b>	truncates resulting binary file to only the number of bytes actually used. Otherwise the resulting outfile is the same size as the romtype given.
<b>-p</b>	presents error conditions as warnings only rather than halting the procedure
<b>outfile</b>	filename of the resulting file produced by <i>ieeeparse</i>
<b>file1</b>	name of first S-record, Intel hex, or binary file for <i>ieeeparse</i> to place in resulting binary file.
<b>base1</b>	first valid address for the ROM being emulated; this setting does not apply to binary source files which are assumed to have the same start address as the base of the emulated ROM
<b>offset1</b>	offset to add to record addresses

### Description

The *ieeeparse* utility converts an IEEE record into a binary image that is loadable into NetROM.

### Compiling source code

*ieeeparse.c* was compiled under SunOS 4.1.x using *gcc* version 2.6.3, using command line:

```
gcc -o ieeeparse -ansi -Wall -pedantic ieeeparse.c
```

The following *gcc* flag is optional:

```
D__USE_FIXED_PROTOTYPES__
```

### **Example**

Convert an IEEE record to a binary format file to be downloaded to NetROM. In the following example the IEEE record is going to be converted into a binary with the name `nr_dptest.bin`. The starting address of the 27c020 that is being emulated in the target is 0xe0040000, so the base is set to that address.

```
ieeeparse -r 27c020 -o nr_dptest.bin nr_dptest.x  
0xe0040000 0x0
```

## rompack

Modifies and/or combines S-record, Intel hex, or binary images into a binary image for loading into NetROM and allows byte swapping in 16, 32, and 64 bit granularity.

### Syntax

```
rompack [-r romtype] [-c romcount] [-f { fillpattern | none }] [-x] [-p] [-o outfile] [-s { 16 | 32 | 64 }] file1 base1 offset1 [file2 base2 offset2 ...]
```

### Options:

<b>romtype</b>	type of ROM image is to be placed in; see page 5-136 for valid values
<b>romcount</b>	number of ROMs of the same <b>romtype</b> that are being used in emulation
<b>fillpattern</b>	pattern to prefill pod memory with before loading images; default is <i>none</i> , which prefills memory with 00
<b>-x</b>	truncates resulting binary file to only the number of bytes actually used. Otherwise the resulting <b>outfile</b> is the same size as the <b>romtype</b> given.
<b>-p</b>	presents error conditions as warnings only rather than halting the procedure
<b>outfile</b>	filename of the resulting file produced by <i>rompack</i>
<b>-s #</b>	swaps the endianness in the image on the given grain size (# is size in bits)
<b>file1</b>	name of first S-record, Intel hex or binary file for <i>rompack</i> to place in resulting binary file.
<b>base1</b>	first valid address for the ROM being emulated; this setting does not apply to binary source files which are assumed to have the same start address as the base of the emulated ROM
<b>offset1</b>	offset to add to record addresses

### Description

The *rompack* utility allows several separate images of varying types, S-record, Intel hex, and binary file formats, to be

merged/ modified at once. This results in a single binary file that can be loaded quickly into NetROM.

### Compiling source code

*rompack.c* was compiled under SunOS 4.1.x using *gcc* version 2.6.3, using command line:

```
gcc -o rompack -ansi -Wall -pedantic rompack.c
```

The following *gcc* flag is optional:

```
D__USE_FIXED_PROTOTYPES__
```

### Examples

Combine an S-record file named *os.s* with a binary image *image.bin*. The binary image will be loaded 0x100 bytes above the starting address of the ROM being emulated, which in this case is 0x0.

```
rompack -r 27c020 -o outfile.bin os.s 0x0 0x0 image.bin  
0x0 0x100
```

Through the constructive use of the base and offset settings an S-record or Intel hex file can be placed at any address in the target. If the target has a 27c020 that starts at address 0xe0040000 and an S-record file that starts at address 0x20000 needs to be placed at the beginning of the ROM being emulated in the target.

```
rompack -r 27c020 -o outfile.bin os.s 0x20000 0x0
```

or

```
rompack -r 27c020 -o outfile.bin os.s 0xe0040000  
0xe0020000
```

Since S-record and Intel hex records have absolute addresses encoded in their format the NetROM needs to know at what base address the ROM being emulated starts. If the ROM that is being emulated starts at 0xe0040000, then the base should be given as 0xe0040000 for these types of files. If the original format of the binary source, in this case, a hex record, is of the

wrong endian, the swap endian option can be used. If the target is a 16-bit target the `-s 16` option can be used as shown here. Also note that the `-f` option has been used here to fill the unused bytes of the resulting output file with `0xff` like in a real PROM.

```
rompack -r 27c020 -f 0xff -s 16 -o outfile.bin  
wrongend.hex 0xe0040000 0x0
```

## thost

Sends an S-record, Intel hex, or binary image from the Unix host to target RAM. Note that *ttarget* must be running on the target for this to work.

### Syntax

```
thost [-s | -i | -b] [-? | -h] [-B base] [-o offset]  
[-v] [-p port] file_name netrom_address
```

### Options

	specifies file format:
<b>-s</b>	S-record
<b>-i</b>	Intel hex
<b>-b</b>	binary default is auto-detection of filetype.
<b>-?</b>	provides help
<b>-h</b>	provides help
<b>base</b>	base address for image; default is 0 for binary images and contained in the image file for S-record and Intel hex
<b>offset</b>	address offset for image; default is 0
<b>-v</b>	verbose mode displays more execution information; verbose mode is not the default
<b>port</b>	NetROM port number to connect to; default port is 1235
<b>file_name</b>	filename of image to be downloaded
<b>netrom_address</b>	network address used by NetROM

### Description

The *thost* utility uses a TCP/IP socket connection between Unix host and NetROM to load code and data images from the specified file to target RAM.

### Compiling source code

*thost.c* was compiled under SunOS 4.1.x using *gcc* version 2.6.3. using the command line:

```
gcc -o turboHost -ansi -Wall -pedantic turboHost.c
```

The following *gcc* flag is optional:

```
-D__USE_FIXED_PROTOTYPES__
```

### Examples

Download Intel hex image **rom.hex** to target RAM with target connected to a NetROM with logical address **netrom22**. Use NetROM port **1235**, the default.

```
thost rom.hex netrom22
```

Download S-record image **rom.srec** to target RAM with target connected to a NetROM with logical address **128.230.1.1**. Use NetROM port **2**. Enable **verbose** mode.

```
thost -v -p 2 rom.hex 128.230.1.1
```



## ttarget

Receives an S-record, Intel hex, or binary image from *thost* on the Unix host. Copies image to target RAM. Note that *thost* must be running on the host for this to work!

### Description

The *ttarget* utility executes in the target at boot-time and uses NetROM's Virtual UART technology to load an image from *thost*. The load address for the downloaded image is a function of the file you are downloading and the *base* and *offset* addresses you provide to *thost*.

### Compiling source code

1. The following instructions assume that you have already included NetROM's Virtual UART communication driver in your target's ROM-based boot code. If you have not, please see Chapter 10.
2. Add *ttarget.c* to the makefile for your target's ROM-based boot image.
3. Edit the function you call to configure NetROM's Virtual UART driver, adding the call to *tload()* just after the call to *nr\_dpConfig()*. See example, below.

### Example

```
#define TURBO 1
#define FLAGVAL 0x1234ABCD /* Arbitrary value */

void configVUART(void)
{
    static uInt32 FirstTimeFlag=0;
    int rc;

    if( FLAGVAL != FirstTimeFlag ) {
        FirstTimeFlag = FLAGVAL;

        rc = nr_ConfigDP( (uInt32)DP_BASE, ROMWORDWIDTH,
                          POD_0_INDEX );

#ifdef TURBO
        tload();
#endif
    }
}
```



```
    } /* if ( TRUE == FirstTimeFlag) */  
  
    nr_SetBlockIO(CHANNEL, True );  
}      /* configVUART */
```

## upload

Sends a binary image from NetROM overlay memory to Unix host.

### Syntax

```
upload [-g podgroup] [-o outfile] netrom_address
```

### Options

**podgroup** pod group to upload; default is podgroup 0  
**outfile** file to save image to; default is outfile.bin  
**netrom\_address** network address used by NetROM

### Description

The *upload* utility uses a TCP/IP socket connection between host and NetROM to upload code and data images from NetROM to a user-specified file on the host computer.

### Compiling source code

*upload.c* was compiled under SunOS 4.1.x using *gcc* version 2.6.3, using command line:

```
gcc -o upload -ansi -Wall -pedantic upload.c
```

The following *gcc* flag is optional:

```
-D__USE_FIXED_PROTOTYPES__
```

### Examples

Upload image to a binary file named **upload.bin** from **podgroup 0** of a NetROM with logical address **netrom22**.

```
upload netrom22
```

Upload image to a file named **rom.bin** from **podgroup 1** of a NetROM with IP address **128.9.233.5**.

```
upload -g 1 -o rom.bin 128.9.233.5
```

## Chapter 7

# 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:

- It does not require that the host system running the debugger user interface have a serial port.
- 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.
- It 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 two 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 a mailbox. Details of the protocol are given in Chapter 9.

---

## 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 capability is built into nearly all popular debuggers. 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 *tgtrreset* 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 side 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 8**

# **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. These are:

- Non-TELNET terminal sessions; documented in this chapter.
- Non-TFTP file downloads; see “download” on page 6-3.
- Uploading emulation memory, see “upload” on page 6-14.



---

## 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.

## Chapter 9

# Emulation Memory Mailbox Protocol

---

NetROM provides a memory mailbox communication protocol for target-to-host communication. Potentially this virtual UART can be very fast, since it does 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.

For targets which can write to their ROM area, NetROM provides a read-write protocol. For targets which cannot write to their ROM area, NetROM provides a *read-read* mechanism which uses a special sequence of reads to cause writes to overlay memory. This chapter describes in detail the protocol used between NetROM and the target system. The API is the same for both read-write and read-read protocols. The target-side driver for this protocol is shipped with NetROM, on the *Drivers & Utilities* diskette.

See Appendix F for a discussion of mailbox protocol implementation issues.

---

## Communication driver API

The full source code for this API is contained on the *Drivers & Utilities Diskette* which shipped with your NetROM unit. This section describes the entry points available to the user of the driver.

The communication driver 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. Table 9-1 lists the routines provided in the driver.

**Table 9-1** Communication driver routines

<b>Routine</b>	<b>Description</b>
<code>nr_ConfigDP</code>	Initializes control structures and configures the target to use the dualport communication protocol.
<code>nr_SetBlockIO</code>	When not in blocking mode, the interface routines merely poll for data and return if none is present. Otherwise they will wait for data to appear.
<code>nr_ChanReady</code>	Returns 1 if NetROM is ready to process messages; 0 otherwise.
<code>nr_Putch</code>	Sends a character to NetROM using the dualport protocol. Actually, it stores characters until the message structure is full or <code>nr_FlushTX()</code> is called. This reduces protocol overhead.
<code>nr_FlushTX</code>	Sends any characters which have been stored but not yet passed to NetROM. Used only with <code>nr_Putch</code> .
<code>nr_Poll</code>	Determines if there is an unread character to read in the channel.
<code>nr_Getch</code>	Reads a character from NetROM, if one is present.
<code>nr_GetMsg</code>	Reads a complete message from dualport memory.



**Table 9-1** Communication driver routines

<b>Routine</b>	<b>Description</b>
<code>nr_PutMsg</code>	Sends a complete message, possibly consisting of several dualport message structures, delineated by the START and END bits in the structures flags fields.
<code>nr_Reset</code>	Requests NetROM to reset the target.
<code>nr_Resync</code>	Requests NetROM to re-initialize its dual port parameters.
<code>nr_SetMem</code>	Requests NetROM to modify the contents of emulation memory.
<code>nr_IntAck</code>	Acknowledges receive interrupt.
<code>nr_Cputs</code>	Puts a user-specified string to the NetROM console; the message won't be transmitted to the host.
<code>nr_TestComm</code>	Determines if NetROM communication is properly configured.
<code>nr_SetEmOffOnWrite</code>	Requests NetROM to turn off emulation memory before modifying memory via the <code>ra_setmem</code> call.

## Porting the driver

Some of the code in the driver needs to be ported to the target system. The following section shows the code which requires porting. The code is from the include file *dpconfig.h*.

The first section of the include file provides target-native storage types. These are used internally to the driver file *dp<sub>target</sub>.c* and the other include file, *dualport.h*; note that *dualport.h* does not require porting to new platforms.

## Driver code

```
/*-----*/
/*          ***  PORT THIS SECTION  ***          */
/*-----*/
#define      ROMSTART      0xFFFF0000L
#define      ROMWORDWIDTH  2
#define      POD_0_INDEX   0
#define      ROMSIZE       RS_27C020

/* MAX_WAIT_FTN_SIZE is used on read-only targets only.
** The wait_ftn() must execute from RAM while NetROM sets Podmem.
** MAX_WAIT_FTN_SIZE is the amount of memory to allocate in RAM
** for the wait_ftn().  nr_ConfigDP() copies wait_fnt() to RAM.
** If too little memory is set aside, nr_ConfigDP() will return
** with an error.  If extra memory is set aside, no problem...
** the nr_ConfigDP() only copies the minimum amount to RAM.
**
** Note: Using an MRI compiler w/ an i960 target,
**
** sizeof(wait_ftn()) = _nr_WaitEnd - _nr_Wait
**                   = 0xE0051560 - 0xE00514F0
**                   = 0xF0
*/
#define MAX_WAIT_FTN_SIZE  0x200

/* The following formula is correct ONLY if dualport is at the default
** location, at the top of pod 0.
** If you move dualport RAM to somewhere else, redefine DP_BASE !!!
*/
/*
#define DP_BASE (ROMSTART + ((ROMSIZE - DUALPORT_SIZE) * ROMWORDWIDTH))
*/

/* The following #define locates DP_BASE at the beginning of Pod 0 */
#define DP_BASE (ROMSTART)

/* Do NOT modify ROMEND */
#define ROMEND ((ROMSTART + ROMSIZE * ROMWORDWIDTH) - 1)

/* If your target CAN write to the memory emulated by NetROM,
** define READONLY_TARGET as False;
** If your target CANNOT write to the memory emulated by NetROM,
** define READONLY_TARGET as True
```



```

*/
#define READONLY_TARGET True

/* Set to True if your target is little-endian, for example, Intel
** Set to False if your target is big-endian, for example, Motorola
*/
#define LITTLE_ENDIAN_TGT True
    /* Big-endian / little-endian conversion routine */

#if(LITTLE_ENDIAN_TGT == True )
    #define swap32(x) \
        (( (long)(x) & 0x000000FF) << 24) + \
        (( (long)(x) & 0x0000FF00) << 8 ) + \
        (( (long)(x) & 0x00FF0000) >> 8 ) + \
        (( (long)(x) & 0xFF000000) >> 24)
    #define swap16(x) \
        (( (int)(x) & 0x00FF) << 8 ) + (( (int)(x) & 0xFF00) >> 8 )
#else
    #define swap32(x) x
    #define swap16(x) x
#endif /* LITTLE_ENDIAN_TGT */

/* Define VETHER only if you are using Virtual Ethernet */
/* #define VETHER */

/* macro to allow other processes to run in a multitasking system */
/* If you are NOT using vxWorks, define YIELD_CPU for your RTOS */
#ifndef vxworks
#include "taskLib.h"
#define nr_YieldCPU() taskDelay(1) /* closest thing in VxWorks */
#else
#define nr_YieldCPU()
#endif

/* Macros to turn interrupts ON and OFF
** Required for nr_SetMem() which MUST be atomic. In multitasking
** systems, unless your code assures that the channel will not be
** written to by another function while nr_SetMem() is running, you'll
** need to define these macros. These macros are
** target-specific, you must supply them.
*/
#define nr_InterruptsOFF()
#define nr_InterruptsON()

```

```

/* The following three macros control target caching.  They are
** required for dualport communication protocol on targets that
** cache the memory region emulated by NetROM.
**
** Cache-related macros:
**   nr_HasCache      Boolean; equals True if your target caches
**                   the memory emulated by NetROM, False otherwise.
**   nr_DataCacheOff Turns caching OFF--you must supply this for your target
**   nr_DataCacheOn  Restores caching--you must supply this for your target
*/
#define nr_HasCache      False /* True */
#define nr_DataCacheOff()
#define nr_DataCacheOn()

```

Three macros (`nr_HasCache`, `nr_DataCacheOff`, and `nr_DataCacheOn`) are provided to control caching. They are required for dualport communications for targets that cache the memory region emulated by NetROM.

The `nr_YieldCPU` macro is provided for target systems which have a non-preemptive operating system which uses system calls to initiate context switches, and which want to use the dualport driver in a blocking mode. Note that blocking mode is disabled by default; to enable blocking, use the `nr_SetBlockIO` function, described below.

The `MAX_WAIT_FTN_SIZE` constant is used by read-only target systems which would like to request that NetROM modify the contents of emulation memory. This macro gives the number of bytes in the `nr_Wait` routine, which is copied into a ram buffer before being executed. The default value for `MAX_WAIT_FTN_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 `nr_Wait` from the link map.



## **Entry points**

This section describes entry points used by the dualport protocol. 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, four channels are supported.



## nr\_ConfigDP

### Prototype

```
Int16  
nr_ConfigDP(uInt32 base, Int16 width, Int16 index)
```

### Parameters

<b>base</b>	first address of dualport RAM
<b>width</b>	width of ROM(s) emulated in bytes
<b>index</b>	location of pod0 within pod group

### Returns

Err_BadLength	if <b>MAX_WAIT_FTN_SIZE</b> is too small to hold <b>nr_Wait()</b>
Err_NoError	otherwise

### 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.

Figure 9-1 shows the interaction of **base**, **width**, and **index** for a variety of target configurations. This routine must be invoked to configure the dualport interface structures before calling any other driver entry points.

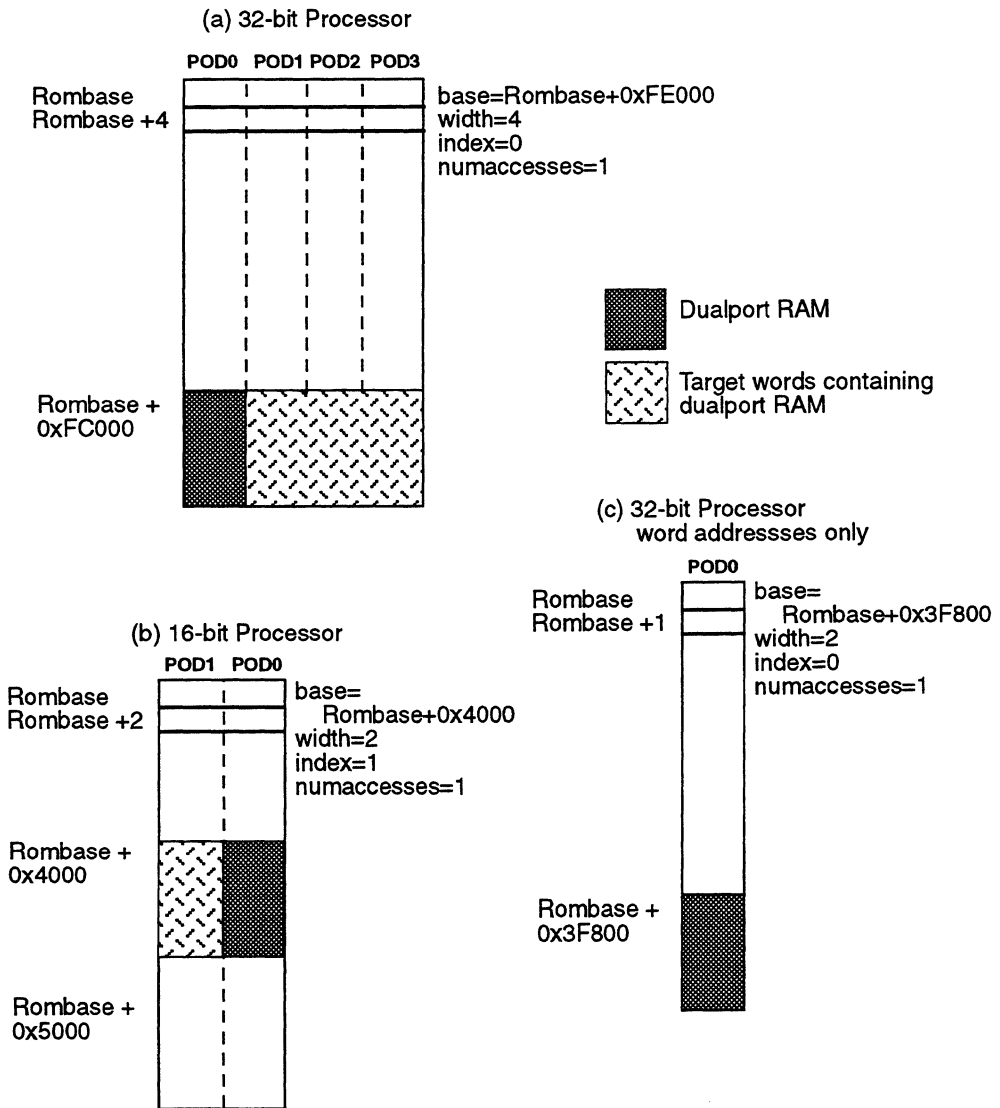


Figure 9-1 Effect of target memory interface in dualport protocol

## nr\_SetBlockIO

### Prototype

Int16

```
nr_SetBlockIO(uInt16 chan, uInt16 val)
```

### Parameters

<b>chan</b>	channel to configure, 0-3
<b>val</b>	false for non-blocking, true for blocking

### Returns

Err_BadChan	if <b>chan</b> is invalid
Err_NoError	otherwise

### Description

When the driver operates in a “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 driver operates in a “blocking” mode routines to get and send characters or messages will repeatedly invoke the `nr_YieldCPU` macro rather than return with an error code. If `nr_YieldCPU` 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 `nr_SetBlockIO()`. The **val** parameter, if true, causes the interface to run in blocking mode; if false, causes the interface to run in non-blocking mode.



## nr\_ChanReady

### Prototype

```
Int16  
nr_ChanReady(uInt16 chan)
```

### Parameters

**chan**                    channel to test, 0-3

### Returns

**Err\_BadChan**            if **chan** is invalid  
**True**                    if the channel is ready  
**False**                   if the channel is not ready

### Description

Call this function immediately after a call to nr\_ConfigDP(). This routine returns **True** 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 **False**, the target should not attempt to perform I/O on the channel. A **False** 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.

## nr\_Putch

### Prototype

```
Int16  
nr_Putch(uInt16 chan, char ch)
```

### Parameters

<b>chan</b>	channel to write to, 0-3
<b>ch</b>	character to send

### Returns

Err_NoError	if the char was sent successfully
Err_WouldBlock	if the char can't be sent
Err_BadChan	if <b>chan</b> is invalid

### Description

This routine sends a character to NetROM using the dualport 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 Err\_WouldBlock. Note that the character will not actually be transmitted until the buffer is full (60 characters) or nr\_FlushTX() is called.



## nr\_FlushTX

### Prototype

```
Int16  
nr_FlushTX(uInt16 chan)
```

### Parameters

**chan**                    channel to flush, 0-3

### Returns

Err\_NoError            if successful  
Err\_BadChan            if **chan** is invalid

### Description

Transmits any chars pending in this channel.

## nr\_Poll

### Prototype

```
Int16  
nr_Poll(uInt16 chan)
```

### Parameters

**chan**                    channel to poll, 0-3

### Returns

True                    if there is a character waiting  
False                   if there is NOT a character waiting  
Err\_BadChan            if **chan** is invalid

### Description

This routine checks to see if a character is waiting to be read from the dualport protocol interface. If so, it returns True; if not, it returns False.



## nr\_Getch

### Prototype

```
Int16  
nr_Getch(uInt16 chan)
```

### Parameters

**chan**                    channel to read from, 0-3

### Returns

Err_NoChar	if channel is non-blocking and there is no char
Err_BadChan	if <b>chan</b> is invalid
char	otherwise

### Description

This routine reads a character from the 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.



## nr\_GetMsg

### Prototype

```
Int16  
nr_GetMsg(uInt16 chan, char* buf, uInt16 len,  
          uInt16* bytesread)
```

### Parameters

<b>chan</b>	channel to read from, 0-3
<b>buf</b>	buffer to read into
<b>len</b>	number of bytes expected, or max bytes for this buffer
<b>bytesread</b>	number of bytes actually received (nr_GetMsg will set this)

### Returns

<b>Err_BadChan</b>	if <b>chan</b> is invalid
<b>GM_NODATA</b>	see description, below
<b>GM_MSGCOMPLETE</b>	see description, below
<b>GM_NOTDONE</b>	see description, below
<b>GM_MSGOVERFLOW</b>	see description, below

### Description

This routine reads a message from the dualport 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 `nr_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, `nr_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, `nr_GetMsg()` will return either `GM_MSGCOMPLETE` or `GM_MSGOVERFLOW`.

Figure 9-2 shows an example of using `nr_GetMsg` in non-blocking mode to receive entire messages.

```

/* reads a message from dualport ram */
int readmsg(chan, buf, lenp)
int chan;
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;
        status = nr_GetMsg(chan, curbuf, bytesleft,
            &bytesread);
        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 9-2** Using nr\_GetMsg to receive a message



## nr\_PutMsg

### Prototype

Int16

```
nr_PutMsg(uInt16 chan, char *buf, uInt16 len)
```

### Parameters

<b>chan</b>	channel to write to, 0-3
<b>buf</b>	buffer to transmit
<b>len</b>	length of buffer to transmit

### Returns

Err_NoError	if the message was sent successfully
Err_NotReady	if NetROM is not ready to process messages
Err_BadChan	if <b>chan</b> is invalid

### Description

This routine sends an entire message to NetROM using the dualport 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.

## nr\_Reset

### Prototype

```
Int16  
nr_Reset(uInt16 chan)
```

### Parameters

**chan**                    channel used for reset message, 0-3

### Returns

Err\_NoError            if the OOB msg was sent successfully  
Err\_NotReady          if NetROM is not ready to process messages  
Err\_BadChan            if **chan** is invalid

### Description

This routine requests NetROM to reset the target.

### Note



---

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

---



## nr\_Resync

### Prototype

```
Int16  
nr_Resync(uInt16 chan)
```

### Parameters

**chan**                    channel to use for resync, 0-3

### Returns

Err\_BadChan            if **chan** is invalid  
Err\_NotReady          if NetROM is not ready  
Err\_NoError           otherwise

### Description

This routine requests the NetROM to re-initialize (resync) the dual port parameters. Call this function after a call to nr\_ConfigDP().

A monitor could use nr\_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.

## nr\_SetMem

### Prototype

```
Int16  
nr_SetMem (uInt16 chan, uInt32 addr, char *buf,  
           uInt16 len)
```

### Parameters

<b>chan</b>	channel to write to, 0-3
<b>addr</b>	start address for write
<b>buf</b>	data to write
<b>len</b>	length of <b>buf</b> ; must be less than 54 bytes

### Returns

Err_NoError	if successful
Err_BadLength	if length is invalid
Err_BadChan	if <b>chan</b> is invalid
Err_BadAddress	if the address is outside ROM

### Description

This routine sends a request to NetROM to modify the contents of emulation memory. The **addr** parameter is the 32-bit address within emulation memory to be modified. The **buf** parameter is a pointer to a buffer from which the data is copied.

It is necessary to run from RAM during part of the set request to avoid potential memory contention problems when NetROM executes the write.



## nr\_IntAck

### Prototype

```
Int16  
nr_IntAck (uInt16 chan)
```

### Parameters

**chan**                    channel to acknowledge interrupt on, 0-3

### Returns

Err\_NoError            if **successful**  
Err\_NotReady           if NetROM is not ready  
Err\_BadChan            if **chan** is invalid

### Description

This routine acknowledges a previous interrupt to the target.  
This routine can help prevent nested interrupts.



## nr\_Cputs

### Prototype

```
Int16  
nr_Cputs(uInt16 chan, char *buf, uInt16 len)
```

### Parameters

**chan**                   channel to write message through, 0-3  
**len**                     length of message, less than 56

### Returns:

**Err\_NoError**           if successful  
**Err\_NotReady**         if NetROM is not ready to process messages  
**Err\_BadChan**          if **chan** is invalid  
**Err\_BadLength**       if the “size” of the buffer is too big

### Description

Puts a string to the NetROM console; the message will not be transmitted to the host.



## nr\_TestComm

### Prototype

```
Int16
nr_TestComm(uInt16 chan, uInt16 lastTest,
            uInt16 endian)
```

### Parameters

**chan** channel to test, 0-3  
**lastTest** See dptarget.c; set to 7 for low-level tests  
**endian** 0 for Intel (little-endian),  
 1 for Motorola (big-endian)

### Returns

**Err\_BadChan** if *chan* is invalid  
**Err\_NoError** otherwise

### Description

This function determines if NetROM dualport communication is working. It first tests the low-level dualport functions `nr_WriteByte`, `nr_ReadByte`, `nr_WriteInt`, `nr_ReadInt`, `nr_WriteBuf`, and `nr_ReadBuf`. After executing the tests, verify their success by typing 'di dpmem 0x50 0x80' at the NetROM prompt. Your NetROM display should look like:

```
0050 6e72 5f57 7269 7465 - 4279 7465 0000 0000 nr_WriteByte....
0060 6e72 5f52 6561 6442 - 7974 6500 0000 0000 nr_ReadByte.....
0070 6e72 5f57 7269 7465 - 496e 7420 0000 0000 nr_WriteInt ....
0080 0000 0000 0000 0000 - 0000 0000 0000 0000 .....
0090 6e72 5f52 6561 6449 - 6e74 2020 0000 0000 nr_ReadInt ....
00a0 6e72 5f57 7269 7465 - 4275 6600 0000 0000 nr_WriteBuf.....
00b0 6e72 5f52 6561 6442 - 7566 2000 0000 0000 nr_ReadBuf .....
```

If the first seven tests fails, either there is a hardware problem, NetROM is misconfigured, or `nr_ConfigDP()` was called with incorrect parameters.

After verifying that the first seven tests passed, you may call `nr_TestComm` with a higher value for the `lastTest` (see `dptarget.c` for test details). Tests eight and higher test `nr_Getch`, `nr_Putch`, `nr_GetMsg`, and `nr_PutMsg`.



## nr\_SetEmOffOnWrite

### Prototype

```
void  
nr_SetEmOffOnWrite(void)
```

### Description

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

### Note



---

Use this routine when you are having trouble with the `nr_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 `nr_SetEmOffOnWrite` before `nr_SetMem()`. The routine needs to be called only one time. A suggested place to call the routine is after the `nr_ConfigDP` function call.

---



## Chapter 10

# General Porting Guide

---

### Introduction

NetROM, a universal debugging platform, integrates the most useful features of hardware-assisted debugging with your software development tools. NetROM works with your debugger and monitor to provide much faster downloads, an Ethernet connection to your target system, the ability to remotely reset your target, ROM emulation, and the ability to set breakpoints in ROM. All of this comes in a package that requires almost no target resources and can be rapidly moved from project to project and from processor to processor. These benefits are summarized in Table 10-1 and Table 10-2.

Table 10-1 Target can write to ROM

Feature	Benefits	Requirements
<b>ROM Substitution</b>	Fast download to ROM	None
	Breakpoints in ROM	None
	Single-step in ROM	None
	Modify ROM contents	None
<b>Run Control</b>	Break/stop	(Possibly) modify monitor's interrupt handler, connect a lead from NR to target
<b>Communication via Virtual UART</b>	Communication with targets lacking serial ports, or frees target serial port, fast download to RAM	Replace UART driver with VUART driver

**Table 10-2** Target cannot write to ROM

<b>Feature</b>	<b>Benefits</b>	<b>Requirements</b>
<b>ROM Substitution</b>	Fast download to ROM	None
	Breakpoints in ROM	Modify monitor's breakpoint routine to call NR <b>setmem</b> function
	Single-step in ROM	Modify monitor's breakpoint routine to call NR <b>setmem</b> function
	Single-step in ROM	Modify monitor's breakpoint routine to call NR <b>setmem</b> function
	Modify ROM contents	Modify monitor's <b>setmem</b> routine to call NR <b>setmem</b> function
<b>Run Control</b>	Break/stop	Modify monitor's interrupt handler, connect a lead from NR to target
<b>Communication via Virtual UART</b>	Communication with targets lacking serial ports, or frees target serial port, fast download to RAM	Replace UART driver with VUART driver

## Porting overview

Porting your target monitor or operating system to NetROM consists of three tasks:

- Replace your target system's serial drivers so that they use NetROM's Virtual UART drivers
- Add NetROM-based run-control to your target software

- If your target cannot write to its ROM area, and you want to debug code in ROM (in NetROM emulation memory), you can add code to your monitor or operation system to request that NetROM change the contents of emulation memory. This is useful for setting breakpoints in “ROM.”

---

## Process

This document shows you how to make the most of your NetROM by describing the steps necessary to enhance target-side monitors and operating systems with NetROM's capabilities. These enhancements are referred to as a NetROM port, and consist of the following steps:

1. Configure your workstation to work with NetROM.
2. Build the monitor without modification and download it to NetROM. Verify operation of the monitor with a serial link before attempting to port to NetROM.
3. Modify the target code so that it uses the NetROM device driver. Through NetROM, the target will communicate with the host-side debugger over Ethernet.
4. (If necessary) modify the monitor to enable breakpoints in ROM. If your target has the ability to write to Flash or EEPROM, you can easily debug code in ROM without modifying the monitor. If your target cannot write to ROM, you can add a function that requests NetROM to set the breakpoint for you.
5. Add remote run control - stopping the target - through NetROM.



---

## Step 1: Configure your host environment for NetROM

---

Note



---

The following steps assume that you are developing on a Unix workstation. If you are using a different operating system, for example Win95, or WinNT, follow the directions that came with your software to add a RARP daemon and a TFTP daemon and configure them for your NetROM.

---

Configuring your host environment for NetROM involves two steps.

- Setting up your RARP daemon for NetROM
- Setting up your TFTP server for NetROM

If you have already established network communications, as described in “Establishing network communications” on page 4-3, go to “Test the monitor without modification” on page 10-6.

➤ **Perform the following steps:**

1. Configure RARP daemon.

Your workstation should invoke the RARP daemon at boot time. To configure it so that it does, check your `/etc/rc.local` file for the following:

```
#
# if /tftpboot exists become a boot server
#
if [ -d /tftpboot ]; then
    rarpd -a;                echo -n ' rarpd'
    rpc.bootparamd;        echo -n ' bootparamd'
fi
```

You will need *root* privileges to create the `/tftpboot` directory.

2. Confirm that the `inetd` daemon is being invoked.  
Check `/etc/rc` or `/etc/rc.local` files for `inetd`.
3. Check that the `/etc/inetd.conf` file invokes the `tftp` daemon.

```
#
# Tftp service is provided primarily for booting.
Most
# sites run this
# only on machines acting as "boot servers."
#
tftp      dgram    udp      wait     root     /usr/etc/
in.tftpd          in.tftpd -s /tftpboot
```

The directory specified after ‘-s’, in this case `/tftpboot`, is where

NetROM’s batch files will be stored. To access files from `/tftpboot`, use NetROM’s `setenv batchpath` command to specify the batchpath as ‘.’ TFTP will automatically prepend `/tftpboot` to the path you specify.

4. Add NetROM’s Ethernet address to `/etc/ethers` file.

NetROM’s Ethernet address is printed on the bottom of the NetROM case. The label will look something like:

```
ETHERNET ADDRESS
00:00:F6:00:4C:E6
```

Add a line like the following to your host computer’s `/etc/ethers` file:

```
00:00:f6:00:4c:e6    netrom1          # netrom for porting work
```

5. Add NetROM’s IP address and hostname to `/etc/hosts`.

Add a line like:

```
128.9.231.57      netrom1          # netrom for porting work
```

---

## Step 2: Test the monitor without modification

The purpose of this step is to verify that NetROM is correctly configured for your target, and that the target monitor and hardware work correctly. For examples showing how to configure your NetROM for your target system, see the NetROM batch files section on page 10-23.

► **Perform the following steps**

1. Turn on your NetROM; your target should be off. Assuming you've completed **Step 1** correctly, when you turn on your NetROM unit, it will use RARP to determine its IP address. You need to create a batch file that configures NetROM for your host and target. Please refer to the Chapter 4, for instructions regarding how to create a batch file. See the example batch files beginning on page 10-23.
2. Once the batch file is ready, telnet into NetROM.
3. From the telnet session, use NetROM's *batch* command to run your newly created batch file.
4. Use NetROM's *newimage* command to download your monitor into NetROM.
5. To verify that you've configured NetROM, your target, and monitor correctly, connect a serial line from the target to your host computer, power on the target, and verify that the monitor is working. You should verify that the target-monitor and host-debugger are working together before moving to **Step 3**.

At this point, you've verified that your host debugger, target monitor, and NetROM are all properly configured and working.

## Step 3: Replace the monitor's serial driver with NetROM's Virtual UART driver

### Overview

The communication driver 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. See the Chapter 8, for details. The Driver API is outlined in Table 10-3.

**Table 10-3** Driver API

Routine	Description
<code>nr_ConfigDP</code>	Initializes control structures and configures the target to use the dualport communication protocol.
<code>nr_SetBlockIO</code>	When not in blocking mode, the interface routines merely poll for data and return if none is present. Otherwise they will wait for data to appear.
<code>nr_ChanReady</code>	Returns <code>True</code> if NetROM is ready to process messages, <code>False</code> otherwise.
<code>nr_Putch</code>	Sends a character to NetROM using the dualport protocol. Actually, it stores characters until the message structure is full - 60 characters - or <code>chan_FlushTX()</code> is called. This reduces protocol overhead.
<code>nr_FlushTX</code>	Sends any characters which have been stored but not yet passed to NetROM. Used only with <code>nr_Putch</code> .
<code>nr_Poll</code>	Determines if a there is an unread character to read in the channel.
<code>nr_Getch</code>	Reads a character from NetROM, if one is present.
<code>nr_GetMsg</code>	Reads a complete message from dualport memory.

**Table 10-3** Driver API (Continued)

<b>Routine</b>	<b>Description</b>
<code>nr_PutMsg</code>	Sends a complete message, possibly consisting of several dualport message structures, delineated by the START and END bits in the structures flags fields.
<code>nr_Reset</code>	Requests NetROM to reset the target.
<code>nr_Resync</code>	Requests NetROM to re-initialize its dual port parameters.
<code>nr_SetMem</code>	Requests NetROM to set memory in emulated ROM.
<code>nr_IntAck</code>	Acknowledges receive interrupt.
<code>nr_Cputs</code>	Puts a user-specified string to NetROM's serial console; the message won't be transmitted to the host. This is a powerful debugging tool that you can use while developing your code.
<code>nr_TestComm</code>	Determines if NetROM communication is properly configured.
<code>nr_SetEmOffOnWrite</code>	Requests NetROM to turn off emulation memory before modifying memory via the <code>ra_setmem</code> call.

## Procedure

To use the VUART, you will either start with a skeleton device driver or modify an existing serial driver. Device drivers typically have four routines that serve as entry points to the driver. These routines

- Configure the driver.
- Call a polling routine to check for characters arriving from the host debugger.
- Call a character-in or message-in routine to read characters from the host debugger.
- Call a character-out or message-out routine to send characters to the host debugger.

After you copy the NetROM driver files into your working directory, you will modify your four driver routines so that they call the appropriate NetROM VUART functions.

► **Copy the files from the *NetROM Drivers & Utilities* diskette**

You will be using the four virtual UART driver files from the *NetROM Drivers and Utilities* diskette. These four files are:

Filename	Description
dptarget.c	C language source code for the virtual UART driver. Do not modify this file.
dpconfig.h	C language include file describing the target system's memory layout. This file <b>will</b> require modification.
dptarget.h	C language include file describing the dualport structure used for communication. Do not modify this file.
dualport.h	C language include file containing macros and data structure definitions. Do not modify this file.

- Copy these files from the diskette to the directory from which you build your device drivers.

► **Modify your monitor's makefile**

1. Replace the serial driver in your makefile with the **VUART** driver file you are creating.
2. Add **dptarget.c** to your makefile.
3. Include the other NetROM routines, in the order shown below, in the VUART driver file:

```
#include "dpconfig.h"
#include "dptarget.h"
#include "dpdualport.h"
```

➤ **Configure the NetROM VUART driver**

- The driver is configured with a call to **nr\_ConfigDP**. Place the call to **nr\_ConfigDP** in your monitor's *init\_communication\_driver* routine.

**nr\_ConfigDP** will also call **nr\_ChanReady** to determine if the channel is ready. If **nr\_ChanReady** indicates that the channel is **not** ready, you should not attempt to use the channel because it is either misconfigured, or the host debugger has not connected. The next call **nr\_ConfigDP** makes will be to **nr\_Resync**. **nr\_Resync** will request that NetROM clear old data from the channel. Without a call to **nr\_Resync**, the host debugger and target monitor will not be synchronized, and communication may or may not take place.

This example driver initialization is from *ppcmon* from SDS.

```
/*
 * Initialize NetROM Dualport protocols
 */
void usr_init(void)
{
/* Use parameters from dpconfig.h, recommended */
nr_ConfigDP( (uInt32)DP_BASE, ROMWORDWIDTH, POD_0_INDEX);
nr_SetBlockIO( CHANNEL, True ); /* I/O blocks; optional */
}
```

Note



---

The parameters passed to **nr\_ConfigDP** are defined in the file **dpconfig.h**.

---

➤ **Configure dpconfig.h for your target system**

- Modify some of the #define statements in the **dpconfig.h** file for proper operation in your target system.

The sections of **dpconfig.h** that you must port to your target are given below, with additional comments interspersed.

**dpconfig.h**

```

/*-----*/
/*          ***      PORT THIS SECTION      ***          */
/*-----*/
#define ROMSTART          0xFFFF0000L
#define ROMWORDWIDTH     2
#define POD_0_INDEX      0
#define ROMSIZE          RS_27C020

...

#define nr_HasCache      False
#define nr_DataCacheOff()
#define nr_DataCacheOn()

```

□ **ROM\_START**

This constant refers to the start address of the ROM being emulated on the target. This address is obtained from a memory map of the target system. For the MVME162 target, the value of **ROM\_START** is 0xFF800000.

□ **ROMWORDWIDTH**

This constant refers to the number of bytes that make up a word on the target system. On the MVME162, for example, one 27c020 (8-bits wide) ROM is being emulated, so **ROMWORDWIDTH** is one. On the CMA277 target, one 27c4096 (16-bits wide) ROM is being emulated, so **ROMWORDWIDTH** is two.

If you are emulating two 8-bit wide ROMs on your target to form a 16-bit word, set **ROMWORDWIDTH** to two. If you are emulating four 8-bit wide ROMs on your target to form a 32-bit word set **ROMWORDWIDTH** to four.

□ **POD\_0\_INDEX**

This constant refers to the index within a target system's word of NetROM's pod number 0. Pod0 on NetROM contains the control logic for implementing the virtual UART. Through the use of the pod order environment variable it is



possible to set Pod0 to any byte in a target's word. To determine the proper value for your target, use the table below. This table assumes a big endian target system.

Target Word Size	ROM size	Pod Order	Pod0_INDEX
32-Bit	8-Bit	0:X:X:X	0
32-Bit	8-Bit	X:0:X:X	1
32-Bit	8-Bit	X:X:0:X	2
32-Bit	8-Bit	X:X:X:0	3
32-Bit	16-Bit	0:X	0
32-Bit	16-Bit	X:0	2
16-Bit	8-Bit	0:X	0
16-Bit	8-Bit	X:0	1
16-Bit	16-Bit	X	0
8-Bit	8-Bit	X	0

(X is *don't care*.)

#### □ ROMSIZE

This constant refers to the size in bytes of the ROM being emulated. In the **dpconfig.h** header file, several common ROM types are defined. An example is **RS\_27C020**.

#### □ HASCACHE

The default for **nr\_HasCache** is **False**. If your target caches the ROM area, set **nr\_HasCache** to **True**. You may need to define the **nr\_CacheON** and **nr\_CacheOFF** macros in the **dpconfig.h** header file.

**dpconfig.h (continued)**

```

/* MAX_WAIT_FTN_SIZE is used on read-only targets only
** The wait_ftn() must execute from RAM while NetROM sets Podmem.
** MAX_WAIT_FTN_SIZE is the amount of memory to allocate in RAM
** for the wait_ftn(). nr_ConfigDP() copies wait_ftn() to RAM.
** If too little memory is set aside, nr_ConfigDP() will return
** with an error. If extra memory is set aside, no problem...
** the nr_ConfigDP() only copies the minimum amount to RAM.
**
** Note: Using an MRI compiler w/ an i960 target,
**
** sizeof(wait_ftn()) = _nr_WaitEnd - _nr_Wait
**                   = 0xE0051560 - 0xE00514F0
**                   = 0xF0
*/
#define MAX_WAIT_FTN_SIZE 0x200

/* The following formula is correct ONLY if dualport is at the default
** location, at the top of pod 0.
** If you move dualport RAM to somewhere else, redefine DP_BASE !!!
*/
#define DP_BASE (ROMSTART + ((ROMSIZE - DUALPORT_SIZE) * ROMWORDWIDTH))

/* The following #define locates DP_BASE to the beginning of Pod 0 */
#define DP_BASE (ROMSTART)

/* Do NOT modify ROMEND */
#define ROMEND ( (ROMSTART + ROMSIZE * ROMWORDWIDTH) - 1)

    □ ROMEND.

        Do not modify.

/* Set to True if your target is little-endian, for example, Intel
** Set to False if your target is big-endian, for example, Motorola
**
*/
#define LITTLE_ENDIAN_TGT True
    /* Big-endian / little-endian conversion routine */

#if(LITTLE_ENDIAN_TGT == True )
    #define swap32(x) \
        (( (long)(x) & 0x000000FF) << 24) + \
        (( (long)(x) & 0x0000FF00) << 8 ) + \

```

```

    (( (long)(x) & 0x00FF0000) >> 8 ) + \
    (( (long)(x) & 0xFF000000) >> 24)
#define swap16(x) \
    (( (int)(x) & 0x00FF) << 8 ) + (( (int)(x) & 0xFF00) >> 8 )
#else
    #define swap32(x) x
    #define swap16(x) x
#endif /* LITTLE_ENDIAN_TGT */

```

#### □ LITTLE\_ENDIAN\_TGT

Specify **LITTLE\_ENDIAN\_TGT True** for little-endian targets; specify **LITTLE\_ENDIAN\_TGT False** for big-endian targets.

```

/* If your target can write to the memory emulated by NetROM,
**   define READONLY_TARGET as False.
/* If your target CANNOT write to the memory emulated by NetROM,
**   define READONLY_TARGET as True.
*/
#define READONLY_TARGET True

```

#### □ READONLY\_TARGET

The default configuration is **True**. This should be used if your target cannot write to its ROMs and you cannot connect NetROM's external write line to a write line on your target. The communication protocol is more efficient when **READONLY\_TARGET** is set to **False**.

```

/* Define VETHER only if you are using Virtual Ethernet */
/* #define VETHER */

/* macro to allow other processes to run in a multitasking system */
/* If you are NOT using vxWorks, define YIELD_CPU for your RTOS */
#ifdef vxworks
#include "taskLib.h"
#define nr_YieldCPU() taskDelay(1) /* closest thing in VxWorks */
#else
#define nr_YieldCPU()
#endif

```

➤ **Call a polling routine to check for characters from the host debugger**

- Modify your character-polling routine to use NetROM functions.

The following is an example of a character-polling routine from the SDS ppcmon monitor.

```
/*
 * Receive Poll routine. Return non-zero if there is a character available.
 */
long usr_poll(void)
{
    return( (long)nr_Poll(CHANNEL) ); /* char ready? */
}
```

Some monitors, for example mon68 from Greenhills, combine the polling function with the character-reading function.

➤ **Call a routine to read characters from the host debugger**

- Modify your character-reading routine to use NetROM functions.

The following example is from the Greenhills mon68 monitor for 68K targets. The INPUT routine has been modified to use NetROM VUART functions. Note that INPUT includes a polling function.

```
/*
 ** Read a character from the serial line or time out if it takes too long.
 */
int INPUT(void)
{
    int i; /* timeout counter */
    int cacrVal = CacheOff68();

    if( nr_Poll(CHANNEL) /* Is there a char? */
        return( nr_Getch(CHANNEL)); /* Use NetROM to read one char */

    CacheOn68(cacrVal); /* restore cache */
    return -1; /* if we timed out, we return a -1 ("EOF") instead */
}
```

The following example is from the SDS ppcmon monitor for PowerPC. `usr_gchar` has been modified to use NetROM VUART functions.

```

/*
 * Read a character from the NetROM communication channel.
 */
long usr_gchar(void)
{
    int rxd;
    rxd = nr_Getch(CHANNEL);
    return( (long)rxd );
}

```

The final example is from a NetROM-based ISI pROBE+ communication driver. `SerialPollConin` has been modified to NetROM VUART functions.

```

/*****
/* SerialPollConin: Get a character from the pROBE+ console          */
/*                                                                    */
/*      INPUTS: None                                                  */
/*      RETURNS: Character from the console.                          */
/*      NOTE: This should only be called after SerialPollConsts()    */
/*            has returned 1 (already know this is a good character) */
/*****
UCHAR SerialPollConin(void)
{
    unsigned int ch = nr_Getch(CHANNEL);

    if( ch == BREAK )
        Brk_Rcvd = TRUE;

    return( (UCHAR)ch );
}

```

- Call a routine to send characters to the host debugger
  - Modify your character-sending routine to use NetROM functions.

The following example is from the SDS ppcmon monitor.

```

/*
 * Transmit the character "outchar".
 */
void usr_pchar(long outchar)
{
    nr_Putch( CHANNEL, (uChar)outchar );
}

```

Note that **nr\_Putch** does not immediately send the characters. Instead they are buffered until either 1) there are 60 characters to send, or 2) **nr\_FlushTX** is called. The SDS monitor includes a routine named **usr\_flush** which will make the required call to **nr\_FlushTX**.

```

/*
 * Flush any characters that usr_pchar placed in the transmit buffer
 */
void usr_flush(void)
{
    nr_FlushTX(CHANNEL);
}

```

Here is a Greenhills mon68 version of the character transmit routine. The call to **nr\_FlushTX** could either be added to the **OUTPUT** routine below, or placed in a function that is called at regular intervals, like the **INPUT** routine. If **nr\_FlushTX** were placed in **INPUT**, you would want to put a count around **nr\_FlushTX** so that it wasn't called every time **INPUT** was called. That would make more efficient use of the Ethernet packets sent by NetROM because they would contain several characters of useful information instead of just one.

```

/*
** Write a character to the serial line.
*/
void OUTPUT(char out)
{
    int cacrVal = CacheOff68();
    nr_Putch(CHANNEL,out);          /* NetROM: Writes one char to VUART */
    CacheOn68(cacrVal);
}

```

**Note**



---

The **INPUT** and **OUTPUT** routines shown here are part of the mon68 BSP. The Greenhills engineers actually chose to implement their NetROM driver as part of the mon68 core. Their implementation offered two advantages over the implementation shown here: 1) the routines they modified were inherently message-based, not character-based, so they made more efficient use of NetROM's transmission and receive mechanisms, and 2) their driver was portable across many targets without modification.

---

The last example for this section is the **SerialPollConout** routine from ISI pROBE+ driver.

```
/* ***** */
/* SerialPollConout: Send a character to the console */
/* */
/* INPUTS: Character to send */
/* RETURNS: None */
/* ***** */
void SerialPollConout(UCHAR c)
{
    nr_Putch(CHANNEL, c );
}
```

---

## Step 4: Add breakpoints in ROM

In targets which can write to the memory overlaid by NetROM, the ability to set emulation memory “comes along for free” with NetROM. These targets include those which program their own flash memory. In targets that do not have a write-line connected to the device which NetROM is emulating, a separate jumper wire can be connected between NetROM and a Write signal on the target. The use of a jumper wire for the Write line only works in targets for which the buffers between the ROM and the CPU allow data to flow in both directions.

If the target is unable to write to the memory emulated by NetROM, the target can request NetROM to write the memory. The function to perform the write is `nr_SetMem`. If the monitor is written to perform all of its writes through one common routine, you modify this routine to call `nr_SetMem` for writes to NetROM. You can use the macro `nr_InROM` to determine if the address is within ROM or not. `nr_SetMem` can handle buffers up to 54 bytes long.

The example below was taken from the SDS `ppcmon` monitor. `setbrk` saves the original instruction, then replaces it with the illegal `brkpt_inst`. Note that `brkpt_inst` is a 32-bit long, and that `brkpt_inst_buf` is a 4-byte character buffer containing the same 4 byte values as `brkpt_inst`.

```
ulong setbrk ( ulong addr )
{
    ulong oldinst;
    int romBreakPoint = 0;

    #if ( READONLY_TARGET == True )
        if( nr_InROM(addr) )
            romBreakPoint = 1;
        else
            romBreakPoint = 0;
    #else
        romBreakPoint = 0;
    #endif
}
```



```

cmd_memaddr = addr;
cmd_recover = CMDR_READ;
oldinst = *((volatile ulong*)addr);

if( 0 == romBreakPoint )
{
    cmd_recover = CMDR_WRITE;
    *((volatile ulong*)addr) = brkpt_inst;
    cache_flush ( addr );
}
else /* New code, using NetROM to set the breakpoint */
{
    cmd_recover = CMDR_WRITE;
    /* brkpt_inst_buf is 4-byte buf containing brkpt_inst */
    nr_SetMem( CHANNEL, addr, brkpt_inst_buf, 4);
    cache_flush ( addr );
}

cmd_recover = CMDR_VERIFY;
if ( *((volatile ulong*)addr) != brkpt_inst ) command_failure();

cmd_recover = 0;
return ( oldinst );
}

```

---

## Step 5: Add remote run control through NetROM

➤ **Add remote run control through NetROM.**

NetROM has eight target control lines. You can use any one of these lines to cause your target to halt execution via interrupts. You can connect another to your target's *reset* line.

The amount of work required to add remote run control via NetROM depends on both your target and the monitor you are using.

- An example of a simple run control enhancement is the 68040 IDP target running the mon68 monitor from Greenhills. One approach is to simply connect a line from **target command** pin 1 on NetROM to the **Abort** button on the IDP board. When the target is running, you can enter the command **set tgtctl 1 on toggle** at the NetROM prompt to stop the target. This does not require any modification of the monitor. Because the monitor is written to handle the NMI generated by the **Abort** button, it behaves correctly when NetROM asserts the NMI.
- Another option, one which does not require connecting the jumper from NetROM to the target, is to tie a polling routine to a timer interrupt. This was the method used by the Greenhills engineers and in the pROBE+ drivers. Simply call a routine every *n* milliseconds that tests the NetROM VUART channel for the receipt of the **Break\_Character**. If the **Break\_Character** was received, halt the user's code.
- The final option is to write an interrupt service routine. Connect the jumper from NetROM to the NMI line on the target - for example the **Abort** button - and write an ISR to handle the interrupt when it is received. This was the approach used with the *ppcmon* monitor. In the case of SingleStep, when the target is running user code, we could assume that the only message the debugger would send would be the **Halt** command. By configuring one of the **target command** pins on NetROM correctly, we were able to assert an interrupt on the target each time a character

was received. We ignore the interrupt while in monitor code, and handle the interrupt when in user code. The command issued on NetROM to set this up is shown in the NetROM batch file for the Cogent603 board, on page 10-25.

---

## NetROM batch files

The first batch configures NetROM for use with a Motorola MVME 162 68040 target. The second batch file configures NetROM for use with a Cogent CMA277 PowerPC603 target. The third batch file configures NetROM for use with a Motorola/Cogent 68040 IDP target.

### Batch file for MVME 162

**begin**

```
# NetROM Setup for MVME 162
# m162.bat
# Batch file for NetROM with MVME162
#
# Command for this batch file: batch m162.bat 192.103.54.226
#
# Note: All lines beginning with the '#' symbol are comments, and you
#       need not type them in. NetROM has online help. Type
#       '?' at the NetROM prompt for a list of topics. For help on a
#       a specific command, type '? command' for example:
#       NRCons>? setenv debugpath
#       will provide help on setting the debugpath.

setenv consolepath serial
setenv debugpath dualport

setenv debugport 2
setenv dprbase 0x3f800
setenv filetype srecord
setenv fillpattern none
setenv groupaddr 0xff800000
setenv groupwrite readonly
#
# 192.103.54.226 is BART...you will want to change this!
#
setenv host 192.103.54.226
setenv loadfile rom.hex
```

```

#
# Note:
#   If you are a running your TFTP server in secure mode (-s option),
#   the server will prepend the TFTP root directory name to the
#   loadpath and batchpath that you set on NetROM.
#
setenv loadpath /
setenv batchpath /
setenv podgroup 0
setenv podorder 0
setenv romcount 1
setenv romtype 27c020
setenv writemode flash
setenv wordsize 8
setenv verify on

#
# Useful aliases
#
alias eon set emulate on
alias eoff set emulate off
alias h history
alias on set debugecho on
alias off set debugecho off
#
# Resync communication channel
#
tgtreset

#
# Load image (optional)
#
# newimage

end

```

## Batch file for Cogent PowerPC 603 (CMA277)

**begin**

```
# Cogent603 Setup
# 2/20/96 by MPH
#
# NetROM command to execute this batch file:
# batch <path/filename> <ip address of your TFTP server>
# batch /a18/sds65/sun_soll/ppcmon/cogmon/nr.bat 128.9.230.1
#
# Note: All lines beginning with the '#' symbol are comments, and you
#       need not type them in. The 'Usage' line and text describing
#       each command below were taken from NetROM's online help. For
#       example, type '? setenv debugpath' at the NetROM prompt

# Usage: setenv debugpath { serial | dualport }
setenv debugpath dualport

# Usage: setenv consolepath { serial | dualport }
setenv consolepath serial

# The path is actually changed when the target is reset using 'tgtreset'
# tgtreset

# Usage: setenv setenv fillpattern { none | value }
setenv fillpattern none

# Usage: setenv host host-addr
setenv host 128.9.230.1

# Usage: setenv loadpath load-file-path
setenv loadpath /a18/sds65/sun_soll/ppcmon/cogmon

# Usage: setenv loadfile filename
setenv loadfile cma277.mot

# Usage: setenv filetype { binary | intelhex | srecord }
setenv filetype srecord

# Usage: setenv batchpath batch-file-path
setenv batchpath /a18/sds65/sun_soll/ppcmon/cogmon
```

```
# Usage: setenv romtype rom-type
```

```
setenv romtype 27c020
```

```
# Usage: setenv podorder podorder-string
```

```
setenv podorder 1:0
```

```
# Usage: setenv target-addr-value
```

```
setenv groupaddr FFF00000
```

```
# Usage: setenv groupwrite {readonly | readwrite}
```

```
setenv groupwrite readonly
```

```
#
```

```
# You can use the 'alias' command to create short abbreviations for
```

```
# common commands
```

```
#
```

```
alias eoff set emulate off
```

```
alias eon set emulate on
```

```
#
```

```
# Typing either 'stop' or 's' at the NetROM prompt pulls NetROM's target  
# control 1 line to ground briefly, then lets it float again. If the line  
# is connected to an interrupt line on the target, and the monitor is set up  
# for "run control," the target code will halt.
```

```
alias stop set tgtctl 1 on toggle
```

```
#
```

```
# dstop enables the SDS debugger to stop the target, for example when the  
# user clicks on the stop-sign or red traffic light icon.
```

```
#
```

```
alias dstop set tgtctl 1 on rx
```

```
dstop
```

```
#
```

```
# Turn target access to emulation memory 'on'
```

```
#
```

```
eon
```

```
#
```

```
# Load the monitor into the target
```

```
#
```

```
newimage
```

```
end
```



## Batch file for Motorola/Cogent 68040 IDP target

**begin**

```
# Moto IDP Setup
# 2/1/96 by MPH
#
# NetROM command to execute this bat file:
# batch /u6/mikeh/gh/idp.bat 128.9.230.1
```

**setenv debugpath dualport**

**setenv fillpattern none**

**setenv host 128.9.230.1**

**#**

```
# Note: In this example, we are using our home directory to store
#       batch files and download images. For this to work, the TFTP
#       server daemon must be configured in non-secure mode.
```

**#**

**setenv loadpath /u6/mikeh/gh**

**setenv loadfile mon68.run**

**setenv filetype srecord**

**setenv batchpath /u6/mikeh/gh**

**setenv romtype 27c020**

**setenv podorder 0**

**setenv groupaddr 0x800000**

**setenv groupwrite readonly**

**alias eoff set emulate off**

**alias eon set emulate on**

**eon**

**end**





## Chapter 11

# Virtual Ethernet

---

Virtual Ethernet is an optional licensed downloadable RAM module for use with NetROM, the Applied Microsystems' embedded-systems development tool. Virtual Ethernet—also called *Vether*—gives target systems the ability to become Ethernet communications devices without requiring that they have Ethernet hardware. This means design engineers can have 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 the Applied Microsystems 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. Vether uses NetROM's shared memory protocols instead of sending and receiving packets through the target Ethernet hardware, and NetROM sends and receives target packets on its Ethernet interface. Figure 11-1 illustrates this process in a logical block diagram.

For additional information about integrating Vether into specific environments, refer to the NetROM application notes on the Applied Microsystems web page (<http://www.amc.com>). If you are not using an environment documented in an application note, use the note as an example that you can adapt to your environment.

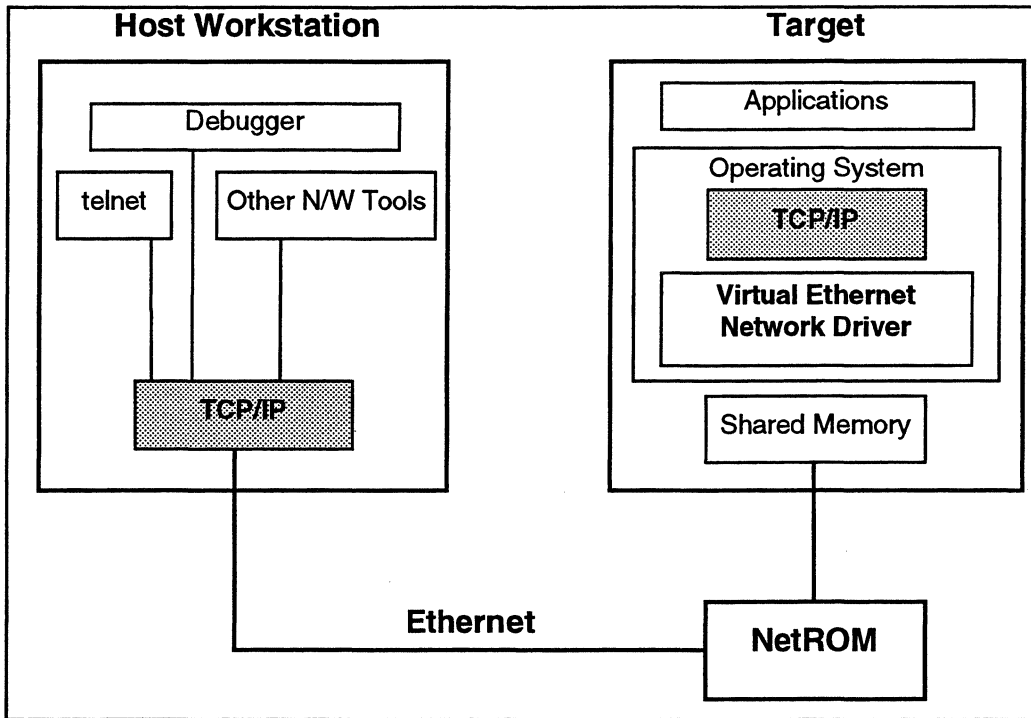


Figure 11-1 Virtual Ethernet logical block diagram

---

## 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 detailed procedures, refer to the application notes on Applied's web page (for example, "*Integrating Virtual Ethernet into VxWorks*").
- 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 to the NIC. For example:  
192.9.1.20 gaia #john smith
- Select an interrupt line on your target (optional).

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 rx0
```

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 rx0
```

Note



---

Only rx0 (the console/debug path dualport channel) can be used by Vether.

---

---

## NetROM setup procedure for virtual Ethernet

Execute the following commands to set up 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 dualport`
- If capable of it, allow target writes to emulation memory:  
`setenv groupwrite readwrite`
- Enable interrupt signaling on packet reception:  
`set tgtctl 1 off rx0`  
The NetROM side of Virtual Ethernet waits for the target's vether driver to be initialized.
- Enable Virtual Ethernet and wait for the target to be initialized:  
`setenv vether on`
- Reset the target. After the target Vether initializes, it synchronizes with NetROM and commences passing packets.

# Appendix A

## Connector Pinouts

---

### RS-232 pinouts

<b>Pin</b>	<b>Description</b>
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

<b>Pin</b>	<b>IEEE 802.3 Signal</b>	<b>Ethernet II Signal</b>
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
chanpath1d	No	Transfers data between the target and the host system.
chanpath2d	No	Transfers data between the target and the host system.
chanpath3d	No	Transfers data between the target and the host system.
Console	No	Provides a user interface on the NetROM Console serial port.
conspatbgd	No	Multiplexes data from the target console to host-side listeners.
debugtld	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
Chan 1	1240	TCP
Chan 2	1241	TCP
Chan 3	1242	TCP
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

#### 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) The Upload Port must be activated before it can be used.



## 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 C0000D2. NetROM now makes several attempts to download its startup file. NetROM will then attempt to download the following startup files: “C0000D2,” then “/tftpboot/C0000D2,” and finally “tftpboot/C0000D2.” 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

---

batchfile	(Hex value of NetROM Ethernet address)
batchpath	/tftpboot
binenv	8
bootflags	rarp bootp autobat
chanpath	serial
chanport 1	1240
chanport 2	1241
chanport 3	1242
consolepath	serial
debugpath	serial
debugport	1235
dprbase	0x3F800
filetype	binary
fillpattern	none
groupaddr	0x00000000
groupwrite	readonly
host	(see Note)
loadfile	image.bin
loadpath	/tftpboot
podorder	0
romcount	1
romgroup	0



### Environment Variables

---

romtype	27c010
verify	on
wordsize	8
writemode	flash

---

**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

---

consecho	off
debugecho	off
emulate	on
udpsrcmode	off





## Appendix F

# Mailbox Protocol Implementation

---

NetROM provides a memory mailbox communication protocol for target-to-host communication. Potentially this virtual UART can be very fast, since it does 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.

This appendix describes implementation issues that were addressed in designing the mailbox communication protocol. See Chapter 9 for information on using the protocol.

---

## 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 F-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.

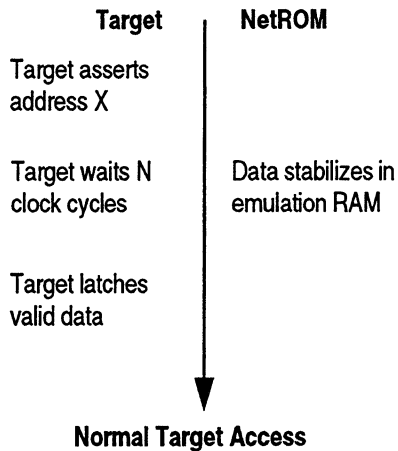


Figure F-1 Memory access cycles

## 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 F-2. This will be referred to as *N-T contention*,



since the NetROM, then the target, attempt to access the same memory during a single memory cycle.

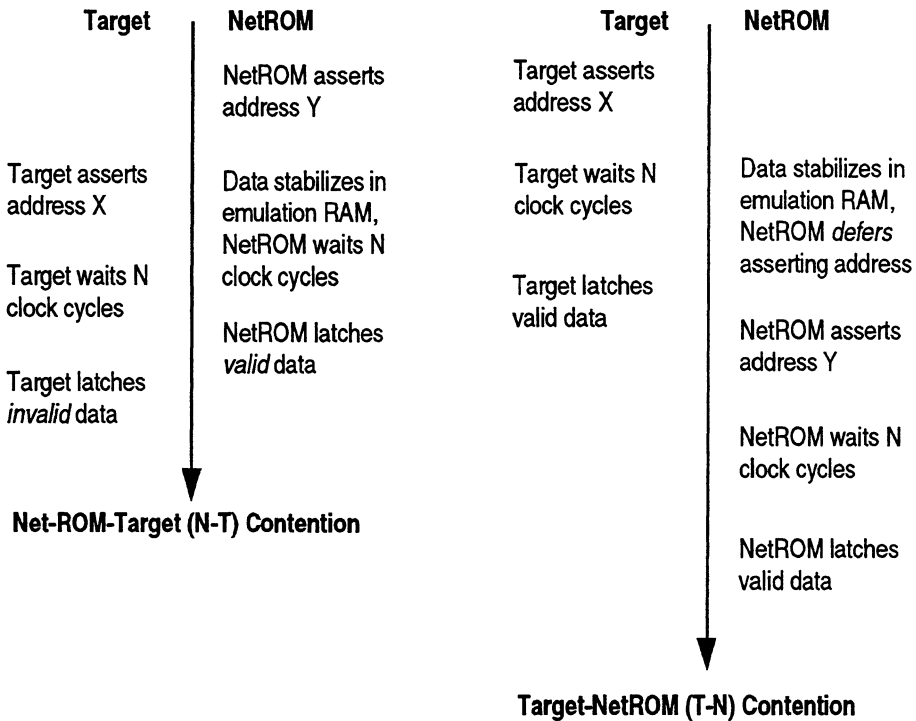


Figure F-2 Contention cycles

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 completed. Unlike the target side, NetROM's hardware is capable of "holding off" the NetROM ROM cycle until the target's cycle 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. This will be referred to as *T-N contention*, since the target, then NetROM, attempt to access the same memory during a single

memory cycle (see Figure F-2). 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

### T-N contention

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 negative 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.

### N-T contention

To address 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 essentially



keeps 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. This protocol is described in detail in subsequent sections.

NetROM's 8192 bytes of dualport RAM are located in emulation pod 0. The dualport 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 8K 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 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 8K emulated by pod 0.

---

## The dualport message structure

Dualport message structures have three parts, a *flags* part, a *size* part, and a *data* part. Each of these parts is of a fixed size, as shown in Figure F-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 another array written by the target.

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 shown in Figure F-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 shown in Figure F-3. Note that the byte containing the READY bit must be written last, after all other bytes of the message are valid.

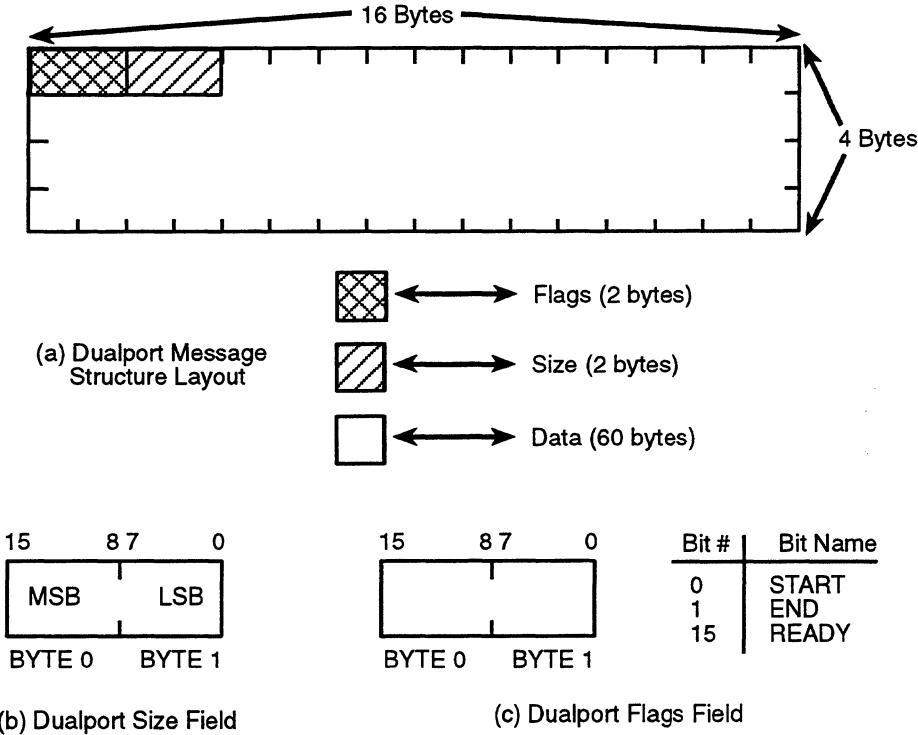


Figure F-3 Dualport message structure

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. The dualport protocol guarantees 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 F-4 depicts the target system reading and verifying the flags field of a dualport message structure.

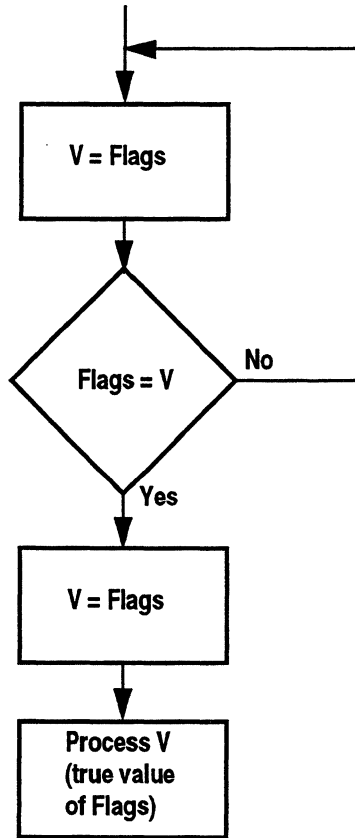


Figure F-4 Target validating dualport flags



## Read-address memory

As 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. It is separate from, but overlaps 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. Second, through a special *read-read* protocol, targets that cannot directly write to emulation memory can write to any address within the dualport region. They do this by performing a sequence of reads to the appropriate addresses.

NetROM supports target systems whose memory interface hardware always “burst reads” from emulation memory. 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. To accommodate such target systems, NetROM can be configured to expect burst reads, and the implementation of the target-side interface driver takes burst reads into account. See the *set raconfig* command on page 5-44 for more information.

---

## Dualport protocol

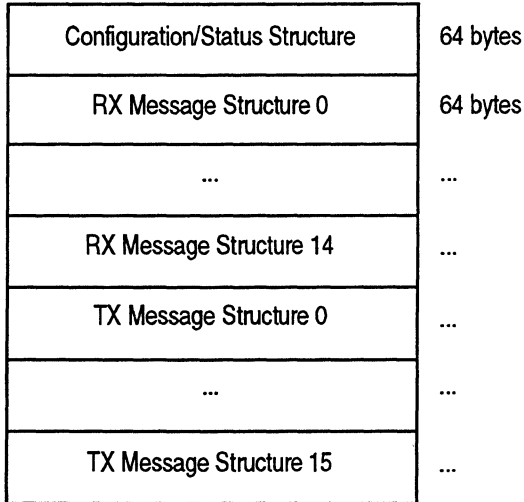
The same dualport protocol is used both on target systems which are capable of writing to their own ROM space and systems that lack this capability. Write-capable systems include those which use FLASH ROMs capable of being reprogrammed by the target system's processor with new images. Targets that cannot directly write to overlay memory use the *read-read* mechanism to write to dualport memory. The dualport protocol allows the target system and NetROM to exchange data in a fashion that amounts to a memory-to-memory transfer between the target system and NetROM's processor.

Figure F-5 shows the layout of dualport memory. Remember that dualport memory may be mapped anywhere within pod 0, and that its default location is at the top 8K of the ROM emulated by pod 0. Note that dualport RAM is divided into eight arrays of message structures—two for each channel. Within each channel, 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 several 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 1 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 dualport RAM), so reading its address sends an interrupt to NetROM.



low address



high address

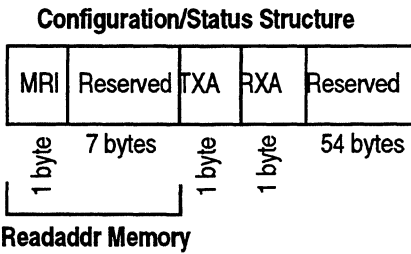


Figure F-5 Dualport RAM channel

## Target-to-NetROM message

The flowchart in Figure F-6 depicts the target system sending a message to NetROM, and Figure F-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 dualport message 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.

## NetROM-to-target message

The flowchart in Figure F-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. The target “polls” the Flags field of the next message buffer. 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.

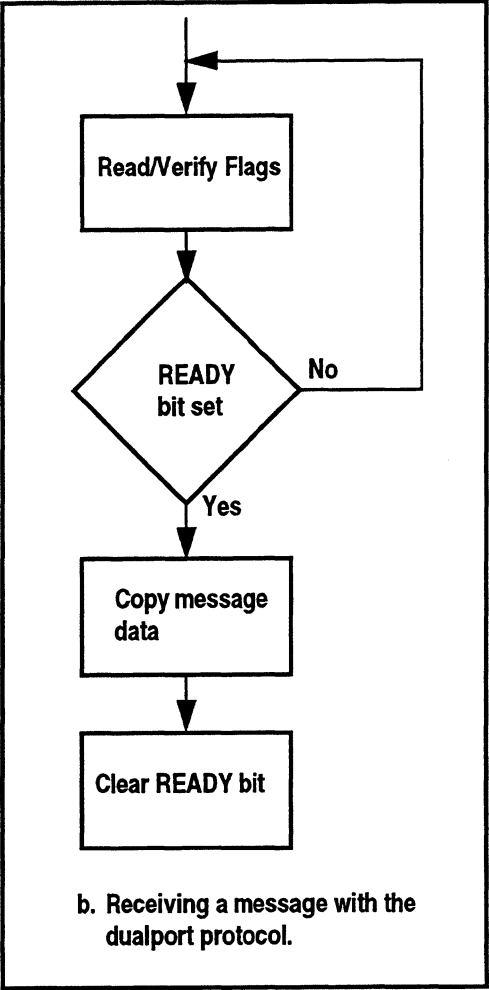
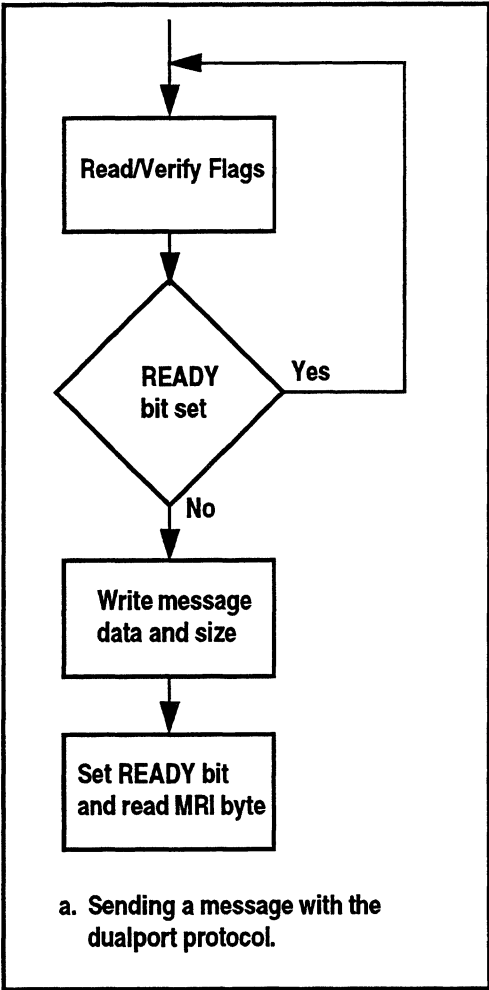


Figure F-6 Dualport protocol



# 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.
Channel Path	A route by which the user communicates with the target.
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.



<b>Dumb Terminal</b>	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.
<b>EPROM</b>	A form of ROM which can be erased and reprogrammed.
<b>Gateway</b>	In IP routing, a gateway is a host on the local network which agrees to route packets to other networks.
<b>Host</b>	A computer on an IP network.
<b>Host, System</b>	A computer which the embedded systems engineer uses to develop code, run debuggers, or establish NetROM sessions.
<b>Internet</b>	A large network of heterogeneous sub-networks.
<b>IP</b>	Internet Protocol, the protocol used to send packets between nodes on the Internet.
<b>IP Address</b>	A 32-bit value, often represented with each byte in decimal and separated by a period.
<b>LED</b>	Light-emitting diode.
<b>Path</b>	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.
<b>PLCC</b>	Plastic Leaded Chip Carrier.
<b>Plug</b>	The end of an emulation cable which is inserted into a ROM socket on the target system.
<b>Pod</b>	Emulation memory on NetROM.
<b>Port</b>	A TCP or UDP identifier, to distinguish between different destinations using the same protocol.
<b>PSOP</b>	Plastic Small Outline Package.

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.
ROM Group	One or more ROMs used together to emulate “words” of ROM memory on the target system.
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 <i>session</i> .
Target System	The hardware whose ROMs are being emulated by NetROM.

<b>TCP</b>	<b>Transmission Control Protocol, a connection-oriented end-to-end transport protocol running on top of IP.</b>
<b>TELNET</b>	<b>A terminal emulation protocol.</b>
<b>TFTP</b>	<b>Trivial File Transfer Protocol, used by NetROM to request files to download.</b>
<b>TSOP</b>	<b>Thin Small Outline Package.</b>
<b>UDP</b>	<b>User Datagram Protocol, a connectionless end-to-end transport protocol running on top of IP.</b>
<b>XON/XOFF</b>	<b>A software handshaking protocol used on RS-232 lines.</b>

# Index

---

## A

active emulation cables 3-8  
address resolution Glossary-1  
    BOOTP 5-16  
    RARP 5-16  
alias 5-88  
arp 5-14, 5-97, Glossary-1

## B

batch 5-89, 5-107  
batch file 5-89, 5-107, 5-124  
    names D-1  
    processing 5-6  
batchfile 5-106  
batchpath 5-107  
binenv 5-108  
boot 5-109  
bootflags 5-109  
BOOTP 2-2, 4-5, Glossary-1  
breakpoint 2-14, 3-29

## C

cables 3-8  
Channel Path Glossary-1  
channel path 2-9  
channel path connection 5-37  
chanpath 5-110  
chanport 5-112  
clear, romset 5-80  
Client Glossary-1  
command line processing 5-2  
command list 5-9  
command signal 2-16, 5-71, E-1  
commands 5-13  
    alias 5-88

arp 5-14, 5-97  
batch 5-6, 5-89, 5-107  
di consecho 5-58  
di debugecho 5-57, 5-59, 5-65  
di dpmem 5-60  
di dpstats 5-61  
di emulate 5-62  
di lanceha 5-63  
di ledmap 5-64  
di lstats 5-66  
di memstats 5-67  
di modules 5-68  
di podmem 5-44, 5-69, 5-135  
di rgconfig 5-70  
di tgtctl 5-71  
di tgtstatus 5-72  
di uart 5-73  
di udpsrcmode 5-74  
di uptime 5-75  
di username 5-76  
di version 5-77  
fill 5-22  
help 5-90  
history 5-5, 5-6, 5-89, 5-91  
ifconfig 5-15  
kill 5-3, 5-33  
ledmap 5-92  
load 5-93  
loadmodule 5-94  
logout 5-95  
netstat 5-17, 5-19  
newimage 5-23, 5-135  
ping 5-3, 5-18, 6-3, 6-5, 6-7, 6-10  
printenv 5-105  
ps 5-2, 5-34  
reset 5-96  
romset? 5-79  
romset clear 5-80

- romset connect 5-81
- romset define 5-82
- romset disconnect 5-83
- romset help 5-79
- romset reset 5-86
- romset show 5-84
- romset slaveaddr 5-85
- route 5-19
- serialcons 5-27
- set consecho 5-37, 5-38
- set debugecho 5-39, 5-41
- set emulate 5-40
- set podmem 5-42
- set prompt 5-43
- set rgconfig 5-44
- set rgname 5-47
- set romupgrade 5-48
- set tgtctl 5-51
- set udpsrcmode 5-53
- set username 5-54
- setenv 5-104
- slip 5-15, 5-20, 5-110, 5-113, 5-115
- stty 5-27, 5-29, 5-98, 8-2
- tgtcons 5-27, 5-29
- tgtreset 5-31
- common entry point
  - nr\_ChanReady 9-11
  - nr\_ConfigDP 9-8
  - nr\_Cputs 9-24
  - nr\_EmOffOnWrite 9-27
  - nr\_FlushTX 9-13
  - nr\_Getch 9-15
  - nr\_GetMsg 9-16
  - nr\_IntAck 9-23
  - nr\_Poll 9-14
  - nr\_Putch 9-12
  - nr\_PutMsg 9-19
  - nr\_Reset 9-20
  - nr\_Resync 9-21
  - nr\_SetBlockIO 9-10
  - nr\_SetMem 9-22
  - nr\_TestComm 9-25
- communication path 2-3
  - channel 2-9

- console 2-7
- debug 2-4, 7-1, 7-2
- download 2-11
- Communications
  - Ethernet 4-3
  - mailbox 9-1, F-1
  - setup 4-3
- configuration 5-70
  - cables 3-8
  - host 5-63
  - information 2-11
  - signal-to-LED mapping 2-17
- configuring servers 4-5
- connect, romset 5-81
- connection
  - AC power 3-5
  - DIP style cables 3-23
  - Ethernet 3-6
  - header style cables 3-24
  - NetROM console 3-27
  - PLCC style cables 3-24
  - surface mount cables 3-25
  - target serial port 3-27
  - write signal 3-29
- consecho, di 5-58
- consecho, set 5-37, 5-38
- console 2-2, 5-98, E-1
- console path 2-7
- console port 5-73
- console serial port 4-7
- console session 5-29
- consolepath 5-113
- Customer support 1-13

## D

- debug control functions 7-3
- debug control port 7-3
- debug path 2-4
- debug path connection 5-39
- debugecho, di 5-57, 5-59, 5-65
- debugecho, set 5-39, 5-41
- debuggers 7-1
- debugpath 5-115

- debugport 5-117
- defaults,setting 4-4
- define, romset 5-82
- di consecho 5-58
- di debugecho 5-57, 5-59, 5-65
- di dpmem 5-60
- di dpstats 5-61
- di emulate 5-62
- di lanceha 5-63
- di ledmap 5-64
- di lstats 5-66
- di memstats 5-67
- di modules 5-68
- di podmem 5-69
- di rgconfig 5-70
- di tgtctl 5-71
- di tgtstatus 5-72
- di uart 5-73
- di udpsrmode 5-74
- di uptime 5-75
- di username 5-76
- di version 5-77
- DIP Glossary-1
- disconnect, romset 5-83
- download 2-2
- download path connection 5-41
- download port C-1
- download utility 6-3
- Downloading non-TFTP files 6-3
- dpconfig.h 9-3
- dpmem, di 5-60
- dprbase 5-118
- dpstats, di 5-61
- drivers and utilities 6-1
- dualport 5-118, 9-18
  - flags F-8
  - memory F-4
  - message structure F-6
  - protocol F-10
- dualport path 2-5, 2-7, 2-9
- dualport RAM 5-27, 5-60

## E

- Electrostatic discharge 1-12
- EMI 1-11
- emulate, di 5-62
- emulate, set 5-40
- emulation cables 3-8
- emulation memory 2-5, 2-11, 5-21, 5-40, 5-44, 5-45, 5-69, 5-96, 5-108, 5-122
- emulation pods 2-11
- environment variables
  - batchfile 5-106
  - batchpath 5-6, 5-89, 5-107
  - binenv 5-108
  - bootflags 5-109
  - chanpath 5-110
  - chanport 5-112
  - consolepath 5-27, 5-113
  - debugpath 5-27, 5-115
  - debugport 5-117
  - dprbase 5-118
  - filetype 5-120
  - fillpattern 5-24, 5-121
  - groupaddr 5-122
  - groupwrite 5-123
  - host 5-89, 5-124
  - loadfile 5-125
  - loading 5-93
  - loadpath 5-126
  - podorder 5-127
  - romcount 5-134
  - romgroup 5-44, 5-135
  - romtype 5-136
  - tgtip 5-139
  - verify 5-140, E-3
  - vether 5-141
  - wordsize 5-142
  - writemode 5-143
- ESD 1-12
- Ethernet 7-1
- Ethernet, virtual 11-1

## **F**

FAX 1-13  
FCC,EMC 1-11  
Filename conventions D-1  
filetype 5-120  
fill 5-22  
fillpattern 5-121  
Firmware 4-3

## **G**

group name 2-14  
groupaddr 5-122  
groupwrite 5-123

## **H**

hardware installation 3-1  
help 5-90  
history 5-91  
host 5-124

## **I**

ieeeparse utility 6-5  
ifconfig 5-15  
Images 2-11  
Installation  
    hardware 3-1  
    software 4-1  
Interference 1-11  
Internet address 1-13  
IP address 2-2, 4-3, 5-12, 5-124, 6-2, D-1,  
    Glossary-2

## **K**

kill 5-33

## **L**

lanceha, di 5-63

LED 5-92, Glossary-2  
LED indicators 4-5  
LED mapping 5-92  
ledmap 5-92  
ledmap, di 5-64  
LEDs 2-18, 2-19  
load 5-93  
loadfile 5-125  
loadmodule 5-94  
loadpath 5-126  
logout 5-95  
longer memory 2-12  
lstats, di 5-66

## **M**

mailbox protocol 5-27, 5-29, 5-110, 5-113,  
    5-115, 9-1, F-1  
memory contention F-2  
memstats, di 5-67  
modules, di 5-68

## **N**

name  
    ROM group 5-47  
NetROM  
    commands 5-9, 5-13  
    Commands (Also, see commands)  
NetROM connections 3-2  
NetROM console 2-2, 3-2  
NetROM reset signal 3-31  
NetROM target serial port 3-27  
netstat 5-17  
network activity LEDs 2-18  
newimage 5-23  
NIS 4-5  
non-TELNET sessions 8-2  
nr\_ChanReady 9-11  
nr\_ConfigDP 9-8  
nr\_Cputs 9-24  
nr\_EmOffOnWrite 9-27  
nr\_FlushTX 9-13

- nr\_Getch 9-15
- nr\_GetMsg 9-16
- nr\_IntAck 9-23
- nr\_Poll 9-14
- nr\_Putch 9-12
- nr\_PutMsg 9-19
- nr\_Reset 9-20
- nr\_Resync 9-21
- nr\_SetBlockIO 9-10
- nr\_SetMem 9-22
- nr\_TestComm 9-25

## **P**

- parallel emulation 3-13
- parallel ROMs 2-12
- passive emulation cables 3-8
- path Glossary-2
  - channel 5-110
  - console 5-29, 5-37, 5-38, 5-113
  - debug 5-27, 5-115
- Phone support 1-13
- ping 5-18, 6-3, 6-5, 6-7, 6-10
- PLCC Glossary-2
- Plug Glossary-2
- pod Glossary-2
- pod order 2-14, 5-70
- podmem, di 5-69
- podmem, set 5-42
- podorder 5-127
- Port Glossary-2
- ports C-1
- printenv 5-105
- processes B-1
- prompt, set 5-43
- protocols C-1
- ps 5-34
- PSOP Glossary-2
- push 4-5

## **R**

- Radio interference 1-11

- RARP 2-2, 4-5, Glossary-3
- RARP file names D-1
- RARP filenames D-1
- read-address memory F-9
- readonly 5-45
- read-only pod groups 2-14
- readwrite 5-45, 5-123
- reset 3-31, 5-96
- reset command signal 5-31
- reset, romset 5-86
- rgconfig, di 5-70
- rgconfig, set 5-44
- rgname, set 5-47
- ROM Glossary-3
- ROM count 2-12
- ROM emulation cables 3-8
- ROM group 5-60, 5-70, 5-118, Glossary-3
  - downloading 2-15
  - name 5-47
  - word width 5-60
- ROM groups 2-12
- ROM type 2-12
- romcount 5-134
- romgroup 5-135
- rompack utility 6-7
- romset clear 5-80
- romset connect 5-81
- romset define 5-82
- romset disconnect 5-83
- romset reset 5-86
- romset show 5-84
- romset slaveaddr 5-85
- romtype 5-136
- romupgrade, set 5-48
- route 5-19, Glossary-3
- RS-232 5-29, Glossary-3

## **S**

- serial emulation 3-13
- serial port 5-110, 5-113, 5-115
- serial ROMs 2-12
- serialcons 5-27
- server Glossary-3



- server configuration 4-5
- session Glossary-3
- set consecho 5-37, 5-38
- set debugecho 5-39, 5-41
- set emulate 5-40
- set podmem 5-42
- set prompt 5-43
- set rgconfig 5-44
- set rgname 5-47
- set romupgrade 5-48
- set tgtctl 5-51
- set udpsrcmode 5-53
- set username 5-54
- setenv 5-104
- show, romset 5-84
- slaveaddr, romset 5-85
- SLIP Glossary-3
- slip 5-20
- SNMP C-1, Glossary-3
- socket Glossary-3
- software
  - installation 4-1
  - setup 4-1
- startup batch file 4-6
- startup batch file,creating 4-4
- Static-sensitivity 1-12
- status LEDs 2-18
- status signal 2-17, 2-18, 5-64, 5-72, 5-92, 5-123
- stty 5-98
- subnet Glossary-3
- subnet mask Glossary-3
- Support 1-13
- system image 5-48

## T

- target 9-7, Glossary-3
- target address 2-15, 5-45, 5-70
- target interface commands 5-21
- target serial port 3-27
- TCP 2-15, 7-2, C-1, Glossary-4
- TCP connection 5-29
- Technical support 1-13

- TELNET 5-29, 8-1, C-1, Glossary-4
- terminal control characters 5-4
- terminal session Glossary-3
- TFTP 2-2, 2-15, 5-107, 5-124, 8-1, Glossary-4
- TFTP server 5-6, 5-89, 5-126
- tgtcons 5-29
- tgtctl, di 5-71
- tgtctl, set 5-51
- tgtip 5-139
- tgtreset 5-31
- tgtstatus, di 5-72
- thost utility 6-10
- TSOP Glossary-4
- ttarget utility 6-12

## U

- UART 5-73
- uart, di 5-73
- UDP Glossary-4
- udpsrcmode, di 5-74
- udpsrcmode, set 5-53
- upload 6-14
- upload port C-1
- upload utility 6-14
- uploading emulation memory 6-14
- uptime, di 5-75
- username, di 5-76
- utilities 6-1
  - download 6-3
  - ieeeparse 6-5
  - rompack 6-7
  - thost 6-10
  - ttarget 6-12
  - upload 6-14

## V

- verify 5-140
- version, di 5-77
- Vether 11-1
- vether 5-141
- virtual Ethernet 11-1

## **W**

Warranty 1-13

wider memory 2-12

word size 2-12, 3-15, 3-16, 5-70, 5-118, 5-142

write signal 3-29

writemode 5-143

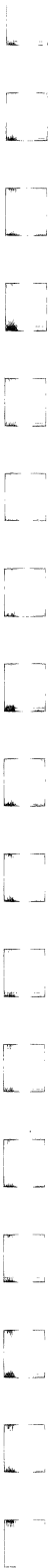
## **X-Y-Z**

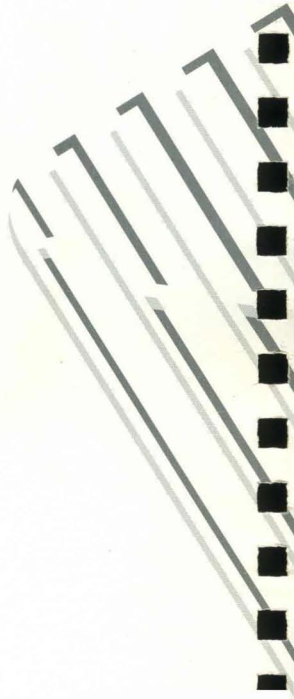
XON/XOFF Glossary-4

Yellow Pages 4-5









Applied Microsystems Corporation