Apple® Macintosh® Coprocessor Platform™ Developer's Guide

Beta Draft, 29 November 1989

Networking & Communications Publications

Network Integration Group

Keith Grigoletto

APPLE COMPUTER, INC.

20525 Mariani Avenue
Cupertino, California 95014
(408) 996-1010

Simultaneously published in the United States and Canada.

Contents

Part I Getting Started With MCP

Part II Software Development /

3 The MCP Software Interface / 3-1

9 A/ROSE Prep / 9-1

10 Using the Forwarder with A/ROSE Prep / 10-1

11 Troubleshooting Guide / 11-1

Part III Hardware Development

Figures and Tables

CHAPTER 1 **What is MCP?**

CHAPTER 2 **Getting Started**

CHAPTER 3 **The MCP Software Interface**

CHAPTER 4 **A/ROSE Primitives**

CHAPTER 5 **A/ROSE Utilities**

CHAPTER 6 **A/ROSE Managers**

CHAPTER 8 **Developing Smart Card Applications**

CHAPTER 9 **A/ROSE Prep**

CHAPTER 10 **Using the Forwarder with Apple IIPC**

CHAPTER 11 **Troubleshooting Guide**

CHAPTER 12 **MCP Card Specifications**

CHAPTER 13 **Listings for the MCP Card**

Preface

This guide helps you create an interface to the Apple® Macintosh® II bus. This guide is written for developers may be within Apple Computer, Inc., as well as third-party developers working under a licensing agreement.

What you should know

You should be familiar with the Macintosh computer and NuBus™. Appendix B lists developer tools, resources, and reference documents that may facilitate your development efforts.

The Macintosh Coprocessor Platform™ (MCP) supports applications written under the Macintosh Programmer's Workshop™ (MPW) development environment, which uses Assembler or C. This guide assumes that you are familiar with MPW and have a working knowledge of MPW C, MPW Assembler, or both.

How to use this guide

The following table provides a road map to information on various subjects of the Macintosh Coprocessor Platform.

MCP Subject:	Location in manual:
General information	***Part I — Getting Started***
What makes up the Macintosh Coprocessor Platform	Chapter 1, "What is MCP?"
Applications or potential uses of MCP	Chapter 1
Installing the MCP card and running a sample program	Chapter 2, "Getting Started"
software specifics	***Part II — Software Development***
A/ROSE™ and A/ROSE Prep software in the Macintosh II family of computers	Chapter 3, "Introduction to the MCP Software"
Task scheduling in the operating system	Chapter 3
Interprocess communication between processes on the Macintosh computer and tasks on the MCP card	Chapter 3 for general information (for additional information, see Chapter 9, "A/ROSE Prep")
Fundamental services of the A/ROSE operating system	Chapter 4, "A/ROSE Primitives"
Library routines available to tasks in your application	Chapter 5, "A/ROSE Utilities"
Operating-system managers that provide services to tasks	Chapter 6, "A/ROSE Managers"

To find out about:	Look in:
Peculiarities of A/ROSE and programming notes (with examples of code)	Chapter 7, "Programming Notes for A/ROSE"
How to develop applications by using MCP software (with examples of code)	Chapter 8, "Developing smart card Applications"
A/ROSE services provided on the Macintosh II	Chapter 9, "A/ROSE Prep"
Forwarding data on an AppleTalk® network system using A/ROSE Prep	Chapter 10, "Using the Forwarder with A/ROSE Prep"
Troubleshooting MCP software	Chapter 11, "Troubleshooting Guide"
The MCPcard NuBus	***Part III — Hardware and Development***
MCP card specifications and information on accessing the NuBus	Chapter 12, "MCP Card Specifications"
PAL listings and parts lists	Chapter 13, "Listings for the MCP Card"
The diagnostics provided for development of the MCP Card	Chapter 14, "Diagnostics for the MCP Card"

Equipment and system requirements

To develop your code, you need the following equipment:

- a NuBus-compatible Macintosh computer running System 6.0.2 or later
- MPW, version 2.0 or later
- one or more MCP cards
- MCP distribution disks
- MPW C and/or MPW Assembler
- the appropriate debugging tools

Connectors and memory requirements are hardware-specific; refer to Part III, "Hardware and NuBus Development", for more information.

Important safety instructions

Before you plug in your Macintosh and get started, read the following important safety instructions.

Conventions used in this guide

Each new term introduced in this book is printed in **bold** type where it is first defined. That lets you know that the term has not been defined earlier, and also indicates that there is an entry for it in the glossary.

Any text displayed in `Courier` typeface is used to represent:

- text that you will see on the screen (such as source code or an example file)
- a command that you enter on the keyboard
- a program or subroutine name
- a parameter or field name

Any text that is surrounded by colons (:) refers to the pathname of a particular folder or file. For example, :A/ROSE:Examples: refers to the folder named "Examples" within the folder named "A/ROSE".

A/ROSE uses C calling conventions, and all registers are preserved except `D0`, `D1`, `A0`, and `A1`. The assembly-language macros also adhere to these conventions.

The following typographic elements mark special mesages to you:

◆ *Note:* Text set off in this manner presents sidelights or interesting points of information.

△ **Important** Text set off in this manner—with the word **Important**—presents important information or instructions. △

△ **Caution** Text set off in this manner—with the word **Caution**—indicates potentially serious problems. Actions could result in system hangs or incompatibility with future versions. △

▲ **Warning** Text set off in this manner—with the word **Warning**—indicates potentially hazardous consequences to you or to your equipment. ▲

Terminology

This document refers to *processes* on the Macintosh computer, and *tasks* under A/ROSE and A/ROSE Prep. A process is an operation or function performed by the Macintosh operating system. A task is a message-driven transaction process that runs on the MCP card. The behavior of a task depends on the messages it receives.

User refers to the end user of the hardware or software product that you will develop by using the Macintosh Coprocessor Platform.

Refer to the glossary at the end of this guide for a comprehensive list of terms and an explanation of each term.

Part I **Getting Started With MCP**

Part I, "Getting Started with MCP," provides:

- an introduction to and overview of the Macintosh Coprocessor Platform
- descriptions of the hardware, software interface, and diagnostics
- instructions for installing the MCP card, operating system, and support software
- a simple "hands-on" exercise that demonstrates how the operating system works with the MCP card

Chapter 1 **What Is MCP?**

THE MACINTOSH COPROCESSOR PLATFORM™ (MCP) is a generic hardware and software foundation to help developers create add-on cards and software applications for NuBus-compatible Macintosh® computers.

Apple Computer, Inc., makes this platform available to assist developers in quickly building Macintosh coprocessor prototypes and to reduce the time-to-market for new products. The Macintosh Coprocessor Platform is available through Apple Computer, Inc. under a licensing agreement. ■

The components of MCP

The Macintosh Coprocessor Platform is made up of hardware and software:

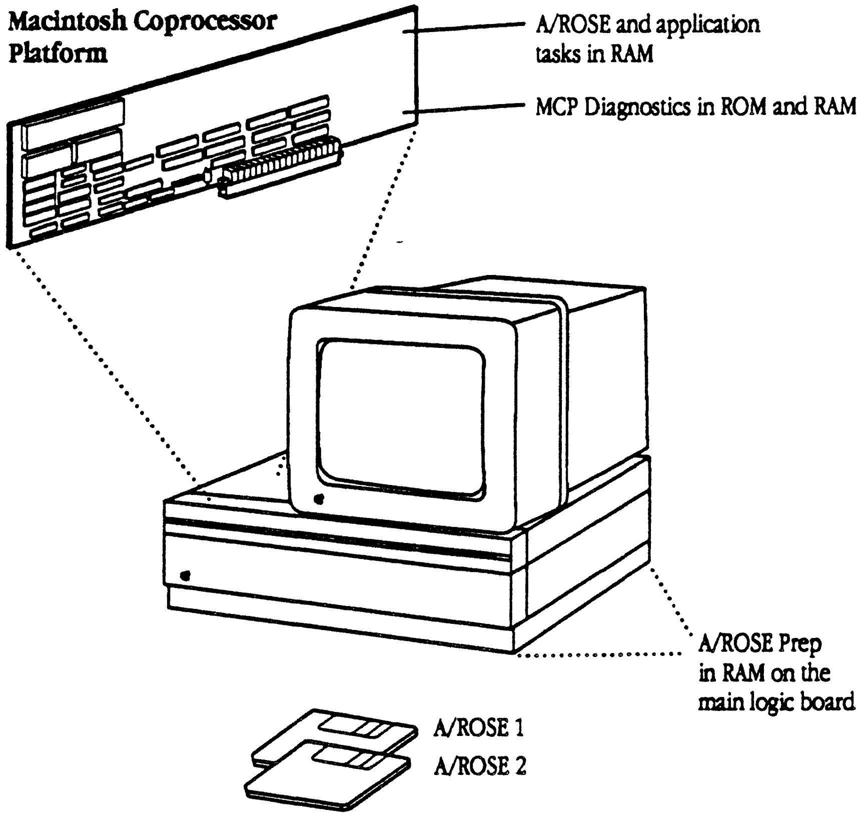
- **Hardware**: the **MCP card**, an intelligent NuBus™ prototype card (such cards may be referred to as smart cards)
- **Software**: two distribution disks (labeled *A/ROSE 1* and *A/ROSE 2*) that include **A/ROSE**™ (Apple Real-time Operating System Environment) and **A/ROSE Prep** (Macintosh II Driver)

  A/ROSE is a multitasking operating system for smart cards, such as the MCP card, and provides an intelligent peripheral-controller interface to NuBus on the Macintosh.

  A/ROSE Prep includes a driver and support software installed in the Macintosh computer. A/ROSE Prep allows Macintosh applications to communicate with an application running under A/ROSE on the MCP card or on another computer.
- **Developmental diagnostic software**: one distribution disk (labeled *MCP_Diagnostic*) that includes the diagnostic application, support code, and examples to test various functions of the MCP–based hardware that you develop

Figure 1-1 shows the MCP software and hardware components for the Macintosh computer.

■ **Figure 1-1** Macintosh Coprocessor Platform for the Macintosh computer

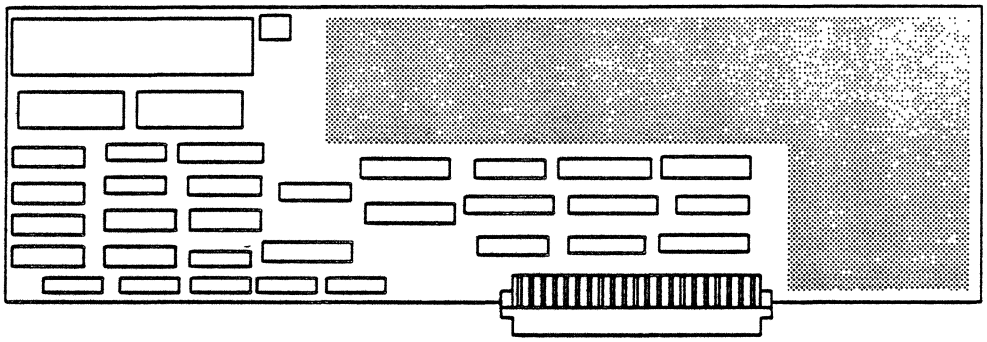

Macintosh Coprocessor Platform
A/ROSE and application tasks in RAM
MCP Diagnostics in ROM and RAM
A/ROSE Prep in RAM on the main logic board
A/ROSE 1
A/ROSE 2

You can customize each of these components, which are described in this chapter, for the particular application or product you want to develop. For more detailed information, refer to Part II, *Software Development or* Part III, *Hardware Developments*.

The MCP hardware

With approximately 26 square inches of space available, the MCP card lets you create a prototype of your application. *Figure 1-2* shows the layout of the MCP card; shading indicates the primary area available for development.

■ **Figure 1-2** The MCP card



The MCP card itself has no input/output (I/O) interface, but is a generic master/slave I/O processor. Affiliated I/O devices that you develop, such as RS-232 ports or token r ing connectors, give the smart card access to the outside world.

The MCP card includes a Motorola 68000 processor operating at 10 megahertz and 512 kilobytes of random access memory (RAM). The NuBus interface provides a bus master interface to NuBus on the Macintosh main logic board. The MCP card acts as a "slot device" to the Macintosh operating system, freeing the processor on the Macintosh to perform other functions.

During development efforts, you may additionally want to use a smart card that is available commercially, such as the AST-ICP (Intelligent Communications Processor) smart card from AST Research, Inc., which includes an I/O interface through four serial ports.

The MCP software

Software for the Macintosh Coprocessor Platform consists of A/ROSE, A/ROSE Prep, and support software (include files, source code examples, and other development software tools). MCP software was created to take advantage of the design features of the MCP card by providing software services to smart card application programs.

The code for A/ROSE and A/ROSE Prep includes a collection of traps, interrupt handlers, and tasks that provide support for task naming, timing services, and intercard and intracard communications using messages. These routines enable a smart card to support a multitasking distributed operating environment for communications and other real-time services on the same card or on other smart cards installed in the Macintosh computer.

◆ *Note:*Card-dependent code has been separated from A/ROSE. The download subroutines will load the appropriate card-dependent code when performing an initial load of A/ROSE operating system to the card.

A/ROSE

A/ROSE provides the operating system and core software services required by MCP cards for on-board applications software. The design of A/ROSE is sufficiently general to support a wide variety of software applications on MCP cards, and offers the functionality described in *Table 1-1*.

■ **Table 1-1** Features of A/ROSE

Feature	Description
Configurability	For maximum flexibility in meeting the needs of a variety of products, large parts of A/ROSE are configurable. A/ROSE code that supports services not required by an application need not be loaded onto the MCP card. To complement configurability, the A/ROSE kernel is as small as possible.
Intercard services	Allows communication between tasks on different cards. Remote system facilities allow allocating and freeing memory, as well as starting and stopping tasks, to support dynamic downloading of tasks on a different smart card in the same machine.
Interprocess communication	Interprocess communication is accomplished through messages that are fixed-size but flexibly formatted. A/ROSE allows dynamic name-binding of tasks to support interprocess communication.
Multitasking	Multiple independent tasks share the CPU on the smart card, under control of A/ROSE. Tasks are always executed in the user mode on the 68000, while interrupt routines and the main program are executed in supervisor mode. This process is important because some 68000 instructions cannot be executed in user mode (such as any instruction that modifies the status register).
Priority scheduling and timer services	Priority scheduling is available to control the order in which tasks use the CPU. A/ROSE supports time slicing and processing that cannot be preempted. Tasks may request one-shot or recurrent notification of time events.
Real-time responsiveness	To deal with the demands of real-time environments, such as communications I/O, both context switching and message passing are designed for very high performance. Memory management is available in an efficient form.

Refer to Part II for more detailed information on A/ROSE and the services it provides.

A/ROSE Prep

A/ROSE Prep is composed of:

- a driver that runs under the Macintosh operating system
- A/ROSE Prep interface code
- library routines (in the file `IPCglue.o`)
- associated support code, including the A/ROSE Prep Name Manager and A/ROSE Prep Echo Manager
- card-dependent routines
- a portion of the download subroutines

The A/ROSE Prep driver handles all message passing (interprocess communication) between processes running under the Macintosh operating system and MCP tasks running under A/ROSE. Periodically, A/ROSE Prep scans for and processes incoming messages, times out slots that have become inactive, and processes outgoing messages. The driver receives messages from and delivers messages to Macintosh processes.

◆ *Note:* Since the Macintosh computer currently does not implement a multitasking operating system, the functions are referred to as *processes* rather than *tasks*.

Refer to Part II for more detailed information on A/ROSE Prep and the services it provides.

Developmental diagnostics

Developmental diagnostics are provided in the firmware. The firmware is provided in the declaration ROM on the MCP card.

These diagnostics are being provided solely as a framework for test verification of board designs. Refer to Part III, *Hardware Development*, for more detailed information.

Developing with MCP

MCP provides hardware and software to assist you in creating

- an application-specific smart card
- Macintosh application software that uses A/ROSE Prep for communication with tasks on the card
- software that executes under A/ROSE on the card

MCP provides a common design to save time in research, design, and development efforts, helping you produce greater and more accurate results in a shorter period of time.

During development, you'll need MPW and standard development tools (linker, C compiler, Assembler, and so forth). The MCP distribution disks provide source code files and examples for A/ROSE and A/ROSE Prep, as well as all of the support software.

You will also need a Macintosh computer with one or more smart cards in the expansion slots. You could conceivably create applications on a Macintosh computer without smart cards installed, and then port it to a Macintosh computer with smart cards installed for testing.

Some of the specific concerns you may have in developing your own application may include the following (refer to the chapters listed for detailed information):

- how to create an A/ROSE or A/ROSE Prep application (Chapter 8)
- how to create interrupt handlers (Chapter 7)
- how to to send data directly to another card (Chapter 5)

The next section describes some development opportunities and potential applications.

Development opportunities and applications

The communications and networking strategy of Apple Computer is to integrate the Macintosh computer into other environments. Some of these environments include those offered by Digital Equipment Corporation (DEC)™, IBM's Systems Network Architecture (SNA), and the proposed Open Systems Integration (OSI) standard.

The on-board operating system provided with MCP gives you the capability to

- off-load tasks usually performed by the central processor, and thus have faster response times (computational speed)
- control and arbitrate multiple communications protocols
- control sessions among users
- run applications in the background

Applications developed with MCP may or may not require users to dedicate a Macintosh computer for the application, depending on how you customize the interface on the card. It is possible to create MCP card applications that, once downloaded, have no dependence on the Macintosh operating system.

Any application or environment that requires the performance of a Macintosh computer can use MCP-developed cards and software. Some of the potential development opportunities described in this section include off-loading task processing, parallel processing, interfacing to or controlling other equipment, data acquisition, and internetworking.

Off-loading task processing

With RAM and a processor on the MCP card, you can off-load a task from the main logic board of the Macintosh and have A/ROSE handle the interprocess communication. A potential development opportunity would be a digital signal processor or a high-speed modem.

Parallel processing

With shared data in a Macintosh computer, the user may want multiple processors to work on data simultaneously. Using multiple cards, an application could

1. Load a task that processes the data onto MCP cards.
2. Send messages to the tasks on the cards with instructions and data.
3. Have the tasks compute in parallel.
4. Receive the results.

Data analysis is an example of this type of an application.

Interfacing or controlling

MCP-developed cards and applications are not strictly a communications interface, but rather a connectivity interface. The product you develop can tie into the Macintosh environment, using the power of the Macintosh to control devices, collect data, or perform some type of analysis. In this situation, the Macintosh computer is dedicated to controlling that device.

Some examples of potential products include

- a numeric controller, machine controller, or any type of device that needs a computerized controller, such as process control in a factory environment (factory automation, specialized devices, or robots)
- medical imaging, such as a system console for a Magnetic Resonance Imaging (MRI) machine

Data acquisition

By developing a SCSI or EDSI (External System Device Interface) connection on the MCP card, you could connect a drive from the Macintosh computer to use it as a database machine distributed over a network, with connections either to or from a host mainframe or other workstations. Examples of applications include instrumentation in a lab, medical applications, or areas in which there is a great deal of testing activity.

Internetworking

The Macintosh Coprocessor Platform offers cost-effective solutions for internetworking needs, including

- providing an environment in which many different kinds of links are simultaneously active
- locally distributing services across networks

- using the intercard communications capability (such as LU 6.2 to EtherTalk)
- using the card as a gateway, bridge, or router into another environment (the other environment may be a nonmainstream environment or a computer that does not use standard protocols)
- enabling other AppleTalk-connected machines to use the communication facilities of the Macintosh

Limitations

When using MCP to develop a NuBus peripheral interface card and associated applications, you are limited in just two aspects:

- what you can program on the card in the existing memory space
- what you can physically build onto the board in the remaining real estate

Chapter 2 Getting Started

THIS CHAPTER shows you how to install the MCP card and software. Then, this chapter takes you through an exercise using the Macintosh Coprocessor Platform card and source-code files. This exercise demonstrates a simple function of the operating system and verifies that the smart card and operating system are working.

This chapter assumes you have already set up your Macintosh II-family computer, according to the instructions in your owner's guide , but have not yet installed any MCP hardware or software. ■

Preparing to use MCP

Before you install the MCP card, follow these steps:

1. Install MPW software on your hard disk into a new folder called MPW.
2. Install Macsbug into the System Folder of your Macintosh.
3. Make a backup copy of the two MCP distribution disks. When you finish copying the disks, remember to put the master disks in a safe place.

One of the MCP distribution disks contains source code and programming examples you will need for application software development and for the exercise in this chapter; the distribution disks include A/ROSE, ROSE Prep, and the support software for both.

◆ *Note:* Please be sure to follow instructions given in, "Installing MCP software," later in this chapter, when copying the contents of the MCP distribution disks to your hard disk. The source code examples check certain locations in the hierarchical file structure for any files needed, not only for the exercise given in this chapter but for all software development efforts.

For a complete guide to the folders and files included on the MCP distribution disks, refer to Appendix A, "Files on the MCP Distribution Disks." (This chapter simply identifies the folders and files you will need for the exercise.)

Now follow the instructions provided in the following sections to install hardware and software for the Macintosh Coprocessor Platform.

Installing the MCP card

This section tells you how to install the MCP card in the Macintosh. If you are not familiar with installing cards, refer to the owner's guide for your Macintosh and to the Preface of this guide for important safety instructions. Follow all instructions and warnings dealing with your system detailed in the owner's guide for your Macintosh.

For your own safety and the safety of your equipment, take the following precautions before installing the MCP card:

- Do not turn on the computer system until you have completed the entire installation process. Turning on the system at the wrong time could result in electrical shock to you or cause damage to your computer system's components.
- Disconnect cables for the monitor, mouse, and keyboard by pulling on the plugs, not the cords. *Leave the power cord plugged in.* The plugged-in power cord acts as a ground for the system, protecting its components from static electrical discharge. Do not defeat the purpose of the grounding plug!

- Touch the power supply case inside the computer to discharge any static electricity that might be on your clothes or body. You can safely touch the power supply if you've just unpacked your computer. However, the power supply can get hot in normal use. If the computer has been on, shut it off and let it cool down for at least five minutes before you open up the main unit and touch the power supply.

To install the MCP card, follow these steps:

1. **Choose the expansion slot in which you would like to install the MCP card.**

   For purposes of this exercise, you can use any slot *except* the second slot from the right of the video card (slot D). However, any slot will work on the Macintosh IIcx. The MCP software downloaded in this example assumes that the MCP card in slot D has an SCC interface; therefore, it is recommended that you use another slot, such as slot B, for this exercise.

   Refer to the owner's guide that came with your particular computer if you need help opening the cover to reach an available expansion slot.

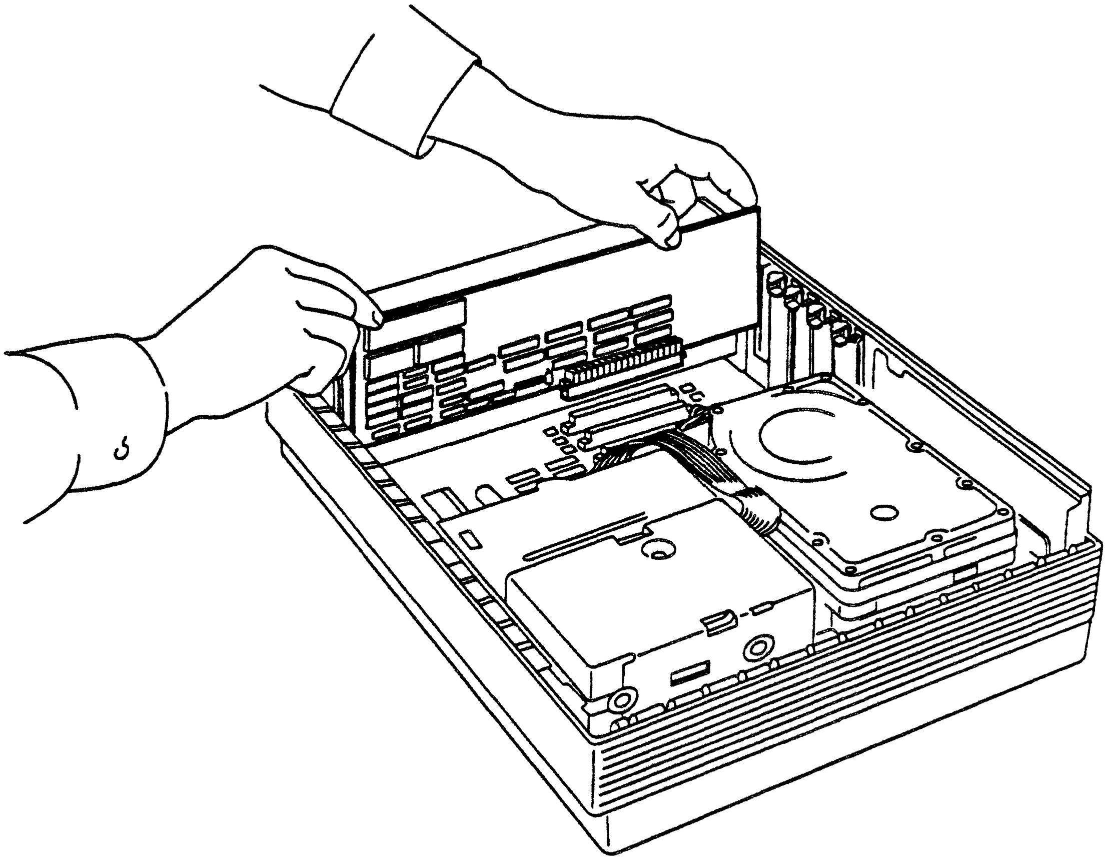
2. **Insert the MCP card into the expansion slot but do not touch the pins on the bottom of the card; handle the MCP card by the top edges only**.

   If your card has a bracket, the expansion cover shield on the card attaches to the inside of the back panel in the same way as the shield you removed in step 1. Just align the card so that the guide fits through the lower slot.

   Align the connector on the bottom of the card, directly over the slot, as shown in *Figure 2-1*.

   Place one hand along the top edge of the card, directly over the connector area, and push down firmly until the connector is fully seated.

■ **Figure 2-1** Aligning the card



△ **Important** Don't force the card. If you meet a lot of resistance, pull the card out and try again.

Don't wiggle the card from side to side when you insert it. Wiggling the card puts unnecessary stress on the card and the slot, and may break electrical connections. △

You can test to see if the card is properly connected by gently trying to lift the card. If it resists and stays in place, it is connected.

3. **If you have purchased other peripheral devices that require cards, install them now.**

   You can use this same method for installing all expansion cards in your Macintosh at any time. Read and follow any instructions that come with other expansion cards you may have. If you plan to install more cards, see Appendix C in the owner's guide to your Macintosh for details on the power available for expansion slots.

4. **Now that the card is installed, reconnect the monitor, the mouse, the keyboard, and plug in any necessary cables.**

If you installed additional cards (such as the AST-ICP smart card) that interface to a network or some other device, connect those cables at this time.

The owner's guide for your Macintosh shows different ways to connect Apple DeskTop Bus™ devices (the keyboard, the mouse, and other devices such as a graphics tablet, a joystick, or another keyboard). You can either daisy-chain them to the keyboard or use one of the back-panel connectors.

◆ *Note: Avoid turning on the power prematurely.* The steps are presented in this order so that the last thing you do is connect the keyboard to a power source. Once the keyboard has power, you could accidentally press the Power On key and turn on your computer before it is appropriate.

5. **Connect any other equipment you plan to use, such as a printer, external disk drive, or modem.**

You will find instructions for connecting those devices in the manuals that came with them. If you're using an external device of any kind that uses a SCSI (Small Computer System Interface) connector, you must connect that device to the one SCSI port on the back of the Macintosh.

▲ **Warning** Connecting a SCSI device to the wrong port can damage your system. You can also damage the system if you mistakenly connect a non-SCSI device (with an RS-232 plug, for example) to this port. Read "Adding SCSI Terminators" in Appendix A of the owner's guide to your Macintosh for important instructions about SCSI terminators. ▲

Once you are satisfied that everything is connected properly, arrange the Macintosh components conveniently in your work area. Turn the main unit so that it faces you, and place the monitor where you want it (on top of the main unit is fine). Position the keyboard and mouse where you can reach them comfortably.

Installing MCP software

To install MCP software, reboot your Macintosh and do the following:

1. **Create a new folder called MCP Software on your Macintosh desktop.**
2. **Copy the contents of the MCP distribution disks to the new MCP Software folder.**

   It takes just a couple of minutes to copy all files from the MCP distribution disks.

△ **Important** Because of naming conventions required by A/ROSE, do not change the names of any of the files or folders copied from the distribution disks. Of course, you can create your own names for the hard disk and first-level folder to which you copy the MCP files and folders. △

◆ *Note:* The A/ROSE folder and A/ROSE Prep folder must be at same level within the new folder you just created, because certain items within the `A/ROSE Prep` file use data in the include files in the A/ROSE folder.

Installing the A/ROSE Prep driver

Now that the files and folders for the MCP software are installed on your hard disk, you will need to install the A/ROSE Prep driver into the System Folder on the Macintosh. Here are the steps that you should follow:

1. **Select the** A/ROSE Prep **folder within the new folder you created on the Macintosh desktop.**
2. **Within the** A/ROSE Prep **folder, open the Examples folder and select the** `A/ROSE Prep` **file.**
3. **Copy the** `A/ROSE Prep` **file into the System Folder of the Macintosh.**

◆ *Note:* You can copy the file in one step by holding down the Option key while dragging the `A/ROSE Prep` file into the System Folder.

4. **Reboot the Macintosh.**

The A/ROSE Prep driver is loaded into the system heap during system startup by an INIT31 resource within the A/ROSE Prep file.

Running a sample program

This section describes how to run a sample program that shows the features and functions of the A/ROSE operating system on the MCP card.

To execute this exercise, you must first run MPW. To do so:

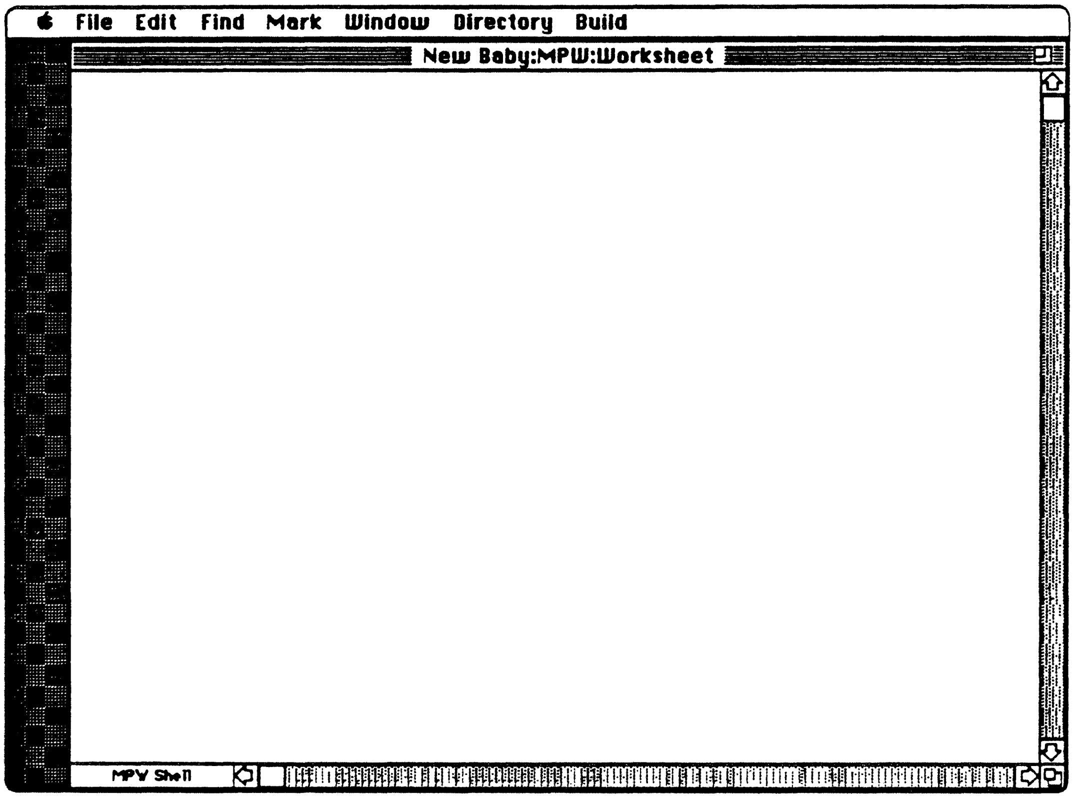
1. **Open the MPW folder.**

   You can open the folder either by selecting it, then selecting Open from the File menu, or by double-clicking the MPW folder icon.

2. **Run MPW by double-clicking on the application called MPW Shell.**

   An MPW worksheet appears, similar to that shown in *Figure 2-2*.

■ **Figure 2-2** MPW window



Selecting files for the sample exercise

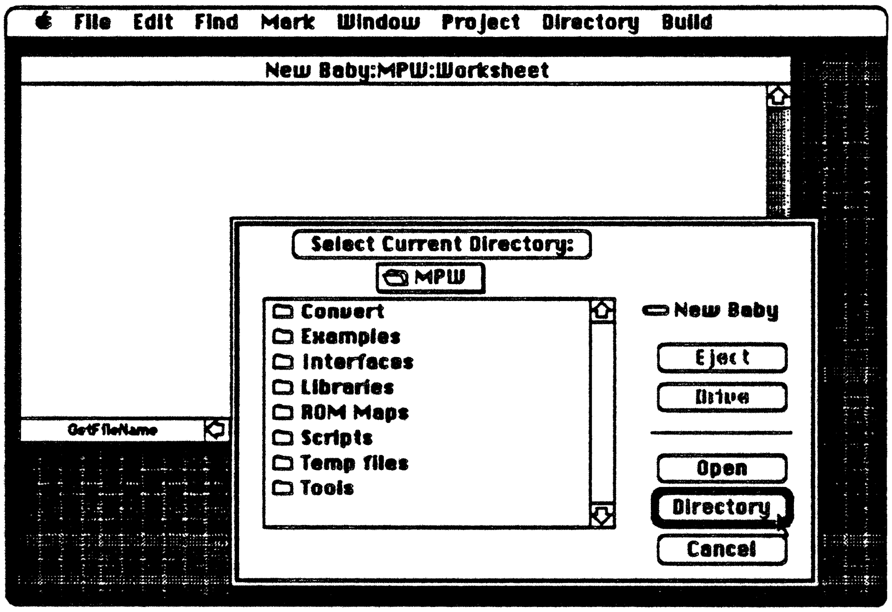
Now you must select the appropriate files to use for the exercise. To do so, first open the folders in which they are located. Follow these steps:

1. **Choose Set Directory... from the Directory menu.**

   A dialog box appears similar to that shown in *Figure 2-3.*

◆ *Note:* The contents of this dialog box will vary depending on the contents of your hard disk.

■ **Figure 2-3** Select Current Directory window



The box beneath the directory title shows all the items in that folder.

2. **Locate and open the folder named MCP Software that you created earlier in this chapter.**

   To open the folder, select the file name, then click Open. You can also open folders and files by double-clicking on the name of the folder you want.

3. **Open the folder named A/ROSE.**
4. **Open the folder named Examples.**
5. **Select the folder named Binaries.**
6. **Click the Directory button.**

   To verify the directory (folder) in which you are working, type the MPW command `directory` and press Enter. To continue the example in this chapter, you should see the following lines on the screen:

```
directory
'New  Baby:MCP  Software:A/ROSE:Examples:Binaries:'
```

where: `directory` is the command you entered to the folder

`'New Baby:MCP Software:A/ROSE:Examples:Binaries:'` is the pathname

◆ *Note:* Your screen will display the pathname and name of the hard disk you are using instead of the text shown in this example.

8. To see the name of the files in the Binaries Examples folder, type the MPW command `files` and press Enter. You should see the following list of all files in the Binaries Examples folders.

```
files
echo.c.o
GenAROSE.c.o
name_tester.c.o
osmain.c.o

ossccint.a.o
pr_manager.c.o
printf.c.o
start
start.map
start.xrf
timeIt.c.o
timer_tester.c.o
trace_manager.c.o
```

For this exercise, you will use the files named `download` and `start`. The download file contains an MPW tool that loads code from A/ROSE to the card; the start file is sample code that runs on the smart card. (Refer to Part II for more detailed information on the download tool.)

Downloading files to the card

To download a file, enter both the command name and the name of the sample file, as follows:

```
'New Baby:MCP Software:A/ROSE:Downloader':download start
```

The `start` file is now running with the A/ROSE operating system on the MCP smart card in your Macintosh. Until you verify that the program is running by using the process described in the next section, you will not see any activity on the screen.

Verifying the sample exercise

Using an MPW tool called the print manager (`pr_manager`), provided on the MCP distribution disks, you can verify that

- the card is running the sample program and file
- communication processes between the card and Macintosh are functioning correctly

The print manager is also designed to run on a card that has an SCC for printing to a terminal (such as an AST-ICP card).

To verify that the program is running, follow these steps:

1. **In the MCP Software folder, find the folder named A/Rose Prep, then the folder named Examples.**

   Follow the steps listed for "Selecting Files for the Sample Exercise," given earlier in this chapter.

2. **Verify the directory using the MPW command** `directory`.

   You should see the following text displayed on the screen:

   ```
   directory
   'New Baby:MCP Software:A/Rose Prep:Examples:'
   ```

3. **Verify the files in that folder using the MPW command** `files`.

You should see the following listing on the screen:

```
files
:DumpTrace:
'A/ROSE Prep'
'AROSE Prep.r'
echo.c
echo_example
echoglobals.a
Makefile
name_tester
name_tester.c
pr_manager
pr_manager.c
RSM_File
RSM_File.c
```

```
RSM_tester
RSM_tester.c
TestR
TestR.c
timeit
timeIt.c
trace_monitor.c
TraceMonitor
--------------------------------------------------------------------------
```

Notice the file for the print manager (named `pr_manager`).

3. To view the activity of the card, type `pr_manager` **and press Enter.**

You'll see messages similar to the following on the screen; for example, the Task Identifier (TID) numbers would be different for different slots.

```
--------------------------------------------------------------------------

pr_manager
Print Manager TID = 4
Starting Main Loop
TID = b00000a - echo tid = b000005
TID = b000008 - Sent message, waiting for reply ----
TID = b000008 - Received msg = FB0706AC, ID = FB002476
TID = b000008 - From: 0364, To: B000008, mCode = -32666, mStatus = -32768
TID = b00000c - RAM test @$fb064898 passed.
TID = b00000c - Testing Slot B
TID = b000008 - About to send msg = FB0706AC, ID = FB0029AC
TID = b000008 - To: 0464, mCode = 102, mDataSize = 1144
TID = b000008 - Sent message, waiting for reply ----
TID = b000008 - Received msg = FB0708F0, ID = FB0029AC
TID = b000008 - From: 0464, To: B000008, mCode = -32666, mStatus = -32768
TID = b000008 - About to send msg = FB070638, ID = FB0029BC
TID = b000008 - To: 0564, mCode = 102, mDataSize = 1144
TID = b000008 - Sent message, waiting for reply ----
TID = b000008 - Received msg = FB0708F0, ID = FB0029BC
TID = b000008 - From: 0564, To: B000008, mCode = -32666, mStatus = -32768
TID = b000008 - About to send msg = FB0706AC, ID = FB0029D1
TID = b000008 - To: 0664, mCode = 102, mDataSize = 1144
TID = b000008 - Sent message, waiting for reply ----
TID = b000008 - Received msg = FB0708F0, ID = FB0029D1
TID = b000008 - From: 0664, To: B000008, mCode = -32666, mStatus = -32768
TID = b000008 - About to send msg = FB07087C, ID = FB0029E1
TID = b000008 - To: 0764, mCode = 102, mDataSize = 1144
TID = b000008 - Sent message, waiting for reply ----
TID = b000008 - Received msg = FB0708F0, ID = FB0029E1
TID = b000008 - From: 0764, To: B000008, mCode = -32666, mStatus = -32768
TID = b000008 - About to send msg = FB070638, ID = FB0029F5
TID = b000008 - To: 0864, mCode = 102, mDataSize = 1144
TID = b000008 - Sent message, waiting for reply ----
TID = b00000c - RAM test @$fb064d18 passed.
TID = b000008 - Received msg = FB0708F0, ID = FB0029F5

--------------------------------------------------------------------------
```

where: `pr_manager` is the command you entered

`Print Manager` is the name of the program that started running under A/ROSE

`TID=4` is the Task Identifier (TID) assigned to that task by A/ROSE

`b0000`*n* is a task (Note that there are several tasks running at the same time.)

These messages originate on the MCP card. This activity not only shows that MCP is functioning correctly, but also displays that multitasking activities are taking place.

The program continues to execute. To stop the activity, press the Command-period key combination. MPW stops the program and displays the following message on the screen:

```
CloseQueue Called
### MPW Shell - pr_manager aborted.
```

You can direct this output as you would do anything else in MPW, such as saving it to a temporary file for printing later.

Where do you go from here?

Now that you've been through a sample exercise, it is time to work on your own applications. Part II, *Software Development*, provides information on software development using A/ROSE and A/ROSE Prep; Part III, *Hardware Development*, provides information on hardware development and diagnostics.

Part II **Software Development**

Part II, "Software Development," provides

- an introduction to and an overview of A/ROSE and A/ROSE Prep (Chapter 3)
- definitions of A/ROSE operating system primitives, utilities, and managers and A/ROSE Prep Services and managers, along with examples in both assembly language and C (Chapters 4, 5, 6)
- information on how to use A/ROSE and A/ROSE Prep (Chapter 7, 9 and 10)
- an exercise to modify standard MCP files to build an application program (Chapter 8)
- programming guidelines and notes for A/ROSE, with program listings for selected examples (Chapter 7)
- a discussion of the Forwarder, an unassociated piece of code that allows A/ROSE tasks to run over AppleTalk.
- a troubleshooting section for crashes and hangs with either A/ROSE or A/ROSE Prep (Chapter 11)

Chapter 3 The MCP Software Interface

SOFTWARE for the Macintosh Coprocessor Platform includes A/ROSE, A/ROSE Prep, and support software (development tools, include files, and examples). This software was created to take advantage of the common design features of the MCP card by providing a common software environment.

This chapter describes the components of the MCP software in greater detail.

What is A/ROSE?

A/ROSE (Apple Real-time Operating System Environment) is a multitasking operating system for smart card devices, such as the MCP card, and provides an intelligent peripheral-controller interface to NuBus.

A/ROSE is a kernel operating system that operates in **supervisor mode** (sometimes referred to as *server mode*). The basic part of the kernel is as small as possible, with the fewest functions necessary to do real work. The design philosophy of the operating system is to not get in the way of what most people want to do; A/ROSE makes minimal assumptions about how things operate. A/ROSE provides basic support services to tasks through system calls **(primitives)** and library routines **(utilities)**.

A/ROSE primitives

A primitive is an A/ROSE system call that provides fundamental services; it is part of the operating system kernel. You must use these services to start and stop tasks, get and free memory, get and free message buffers, send and receive messages, change the scheduling parameters of a task, and set the hardware-interrupt priority level. Refer to Chapter 4 for more detailed information on A/ROSE primitives.

A/ROSE utilities

A *utility* is the library code needed to make the functional call interface between the kernel and other code providing higher-level services (such as the A/ROSE managers or code you develop for other tasks). The utilities allow you to move data, manage buffers, obtain the operating environment, translate NuBus addresses, and register and look up task names through the Name Manager. Refer to Chapter 5 for more information on A/ROSE utilities.

A/ROSE managers

Managers are tasks that carry out higher-level services on behalf of other tasks. A/ROSE managers extend the kernel to provide services that are not in the kernel, but are useful for all users of the A/ROSE operating system.

Managers exist on top of the kernel. Because code for the managers is provided on the MCP distribution disk, you can incorporate desired functions into the application program you develop using appropriate calls. Both managers and application code for tasks that you develop operate in **user mode** (sometimes referred to as *client mode*).

Figure 3-1 shows the relationship between the A/ROSE kernel, primitives, utilities, and managers.

■ **Figure 3-1** Structure of A/ROSE

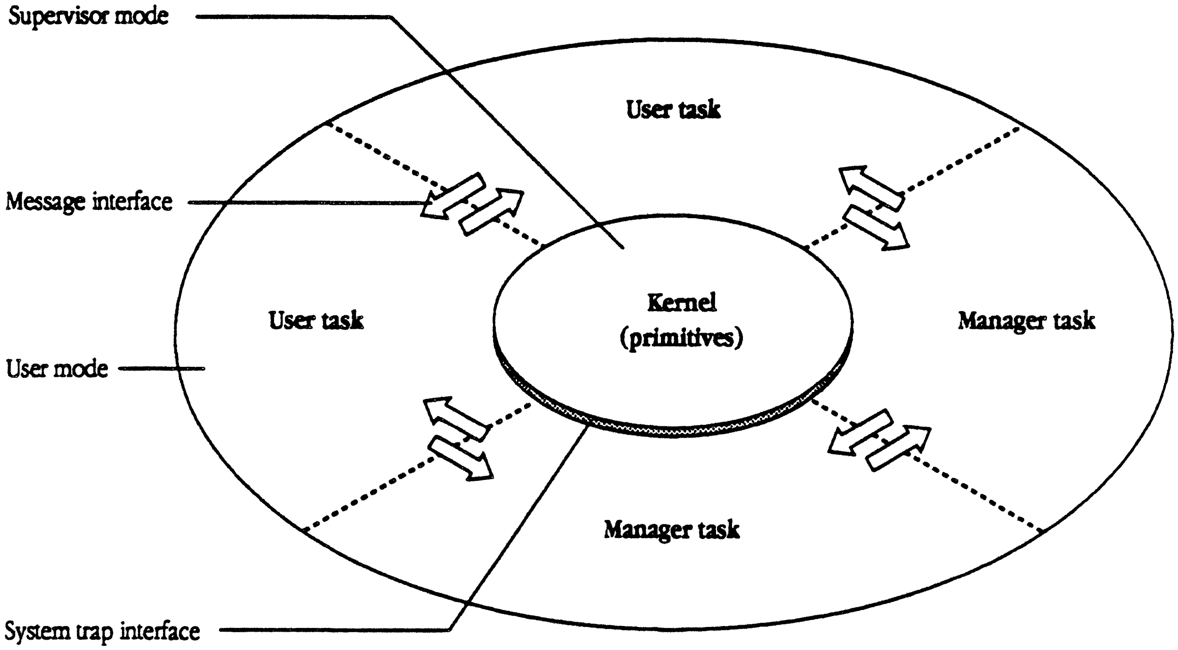
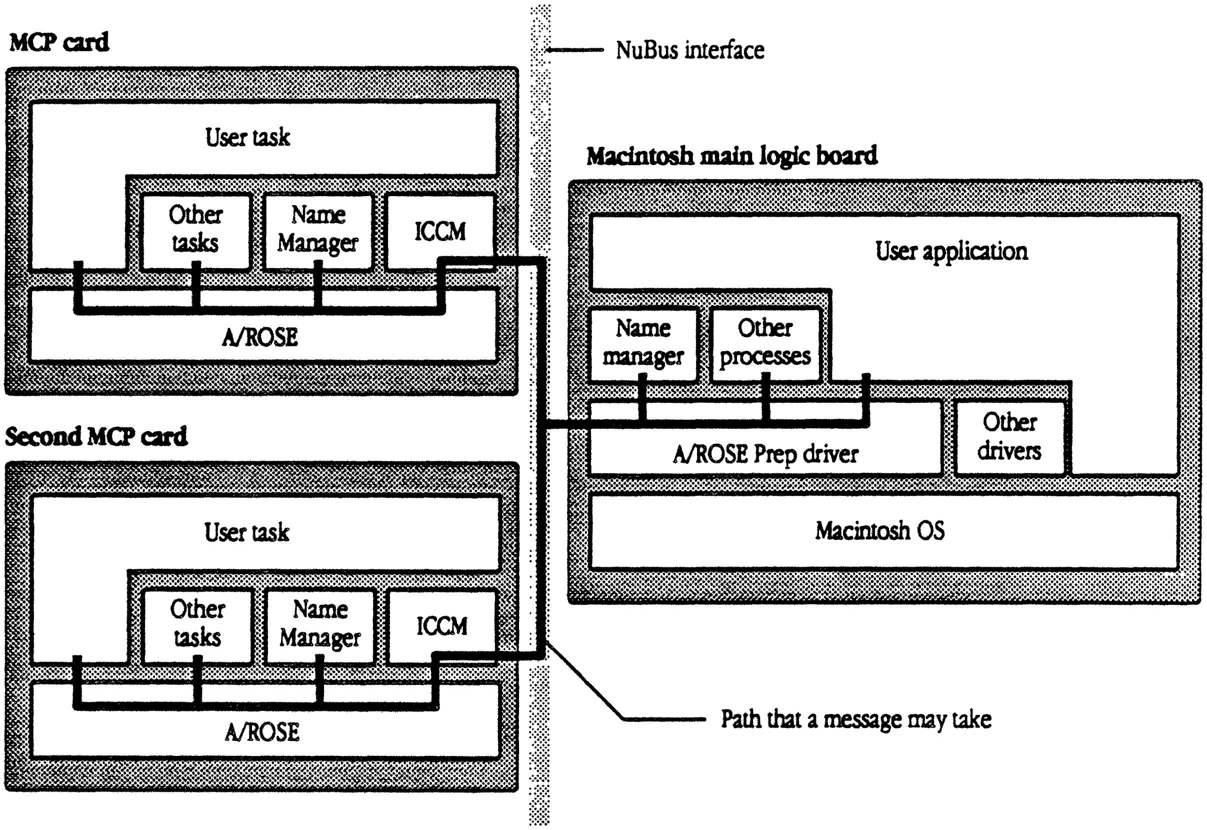


Figure 3-2 illustrates the flow of information between A/ROSE and the managers on an MCP card.

- **Figure 3-2** Flow of information between A/ROSE and managers



This section provides a brief description for each of the A/ROSE managers (refer to Chapter 6 for more detailed information):

- Echo Manager
- InterCard Communications Manager
- Name Manager
- Print Manager
- Remote System Manager
- Timer Manager and Timer Library
- Trace Manager

Echo Manager

The Echo Manager returns each message it receives to the sender. You can use the Echo Manager primarily during the early stages of development for

- sending test messages
- determining the time required for a round-trip message response

InterCard Communications Manager (ICCM)

The InterCard Communications Manager (ICCM) is responsible for sending and receiving all messages between smart cards installed in the same machine. A/ROSE delivers any messages addressed off-card (off the active MCP card) to A/ROSE Prep or ICCM. ICCM forwards the message to a peer ICCM on the destination smart card or A/ROSE Prep on the Macintosh main board for delivery. ICCM also allows tasks to request information about other cards; namely, the tasks ask for information about the existence of a smart card in a given slot and the task identifier of its Name Manager.

Name Manager

The Name Manager allows A/ROSE tasks to find the task IDs of other A/ROSE tasks, given the names of those tasks.

To provide these naming services, the Name Manager allows tasks to

- register and unregister their own name with the Name Manager
- look up the task identifier of named tasks
- look up the name of a task corresponding to a given task identifier
- become visible to other tasks on the same card and, optionally, to tasks on the Macintosh main logic board or other smart cards

The Name Manager supports searching for names using wildcard characters; the Name Manager also provides for notifying tasks of the loss of communication with a smart card or the termination of a task.

The Name Manager operates with a single message loop: for each message it receives, it performs the service specified in the message code. The Name Manager handles errors by indicating the failure status in the message sent back to the requesting task.

Print Manager

The Print Manager is a diagnostic tool that allows you to put print statements in your program and get the output printed on a display. The display can be output either on the Macintosh or to a serial port.

Remote System Manager (RSM)

The Remote System Manager (RSM) provides a mechanism for supporting dynamic downloading of tasks to another smart card in the same machine. RSM provides two types of services:

- getting and freeing memory
- starting and stopping tasks

RSM operates with a single message loop; for each message it receives, it performs the service specified in the message code. For each kind of request message, RSM on the remote (destination) card executes the applicable A/ROSE primitive on behalf of the requesting task. RSM handles errors by indicating the failure status in the message sent back to the requesting task.

Timer library and Timer Manager

The timer library allows user tasks to receive "wake-up" calls and activates timing, cancels timing, sets timing, and so forth. Use the timer library when you want to use periodic timers, for high-performance timers, and when you want to cancel a timer reliably when an event occurs.

The timer library is available in the file `os.o` on the MCP distribution disk. The timer library provides three types of timing services to tasks:

- time-event notification
- time-event query
- time-event cancellation

The user task can request two types of time events:

- one-shot, in which only one time-event notification message is sent
- periodic, in which time-event notifications are sent at specified intervals

The Timer Manager is provided with this version of the A/ROSE software for compatibility with previous versions; its function has been replaced by the Timer Library.

Trace Manager

The Trace Manager provides a way to dynamically trace all the message exchanges in the operating system. The Trace Manager can be an extremely useful debugging facility; when all else fails, you can trace messages and slow the process down in order to see things you could not see before. The Trace Manager traces everything except itself: every message that is sent is put in a log file.

△ **Caution** A limitation of using the Trace Manager is that it alters time where a program is concerned, and therefore may affect the operation of a task if timing is a factor. Therefore, some operations work while others do not when the Trace Manager is running.

For example, the Trace Manager may impact programs that control high-speed I/O devices. Because messages are traced, they may not return fast enough to activate the device, or the timing may be altered. This results in errors that are time-dependent. △

What is A/ROSE Prep?

A/ROSE Prep is a combination of a driver and support software found in the A/ROSE Prep file in the A/ROSE Prep folder on the MCP distribution disk.

A/ROSE Prep provides message-passing and naming services for communication among the Macintosh, tasks on the Macintosh, and tasks on smart cards. Interprocess communication is accomplished through messages that are fixed-size but flexibly formatted. (A/ROSE Prep provides functionality similar to the InterCard Communications Manager on A/ROSE).

◆ *Note:* This document refers to *processes* on the Macintosh, and *tasks* under A/ROSE and A/ROSE Prep.

An application that uses A/ROSE Prep must have an initial call to `OpenQueue` to establish its use of A/ROSE Prep. Messages are sent and received via the `Send` and `Receive` calls, much like tasks under A/ROSE. Several source-language examples of applications are provided in the A/ROSE Prep:Examples folder on the MCP distribution disk. Refer to Chapter 9 for a more detailed description of the services provided by A/ROSE Prep.

A/ROSE Prep driver

A/ROSE Prep services are handled by the A/ROSE Prep driver, which handles all message passing between processes running under the Macintosh operating system and A/ROSE tasks on the smart card over the NuBus. Using calls to the A/ROSE Prep driver, the Macintosh process sends messages to and receives messages from tasks on the smart card and on processes on the Macintosh.

You will need to place the A/ROSE Prep file in the System Folder; routines contained in the file are installed by the INIT31 mechanism during system startup. (Refer to Chapter 2, "Getting Started," for installation instructions.)

During initialization, the driver sets up a communication area, and then searches NuBus slots for the ICCM communication areas of smart cards installed in the Macintosh, much as the A/ROSE ICCM does. For each valid ICCM communication area found, the driver stores the address of the A/ROSE Prep communication area in a vector in the ICCM's communication area.

Periodically, A/ROSE Prep scans for `Receive` operations that have timed out, incoming messages, active slots that have timed out, and outgoing messages. The driver receives messages from and delivers messages to the Macintosh processes.

A/ROSE Prep library

The interface between a Macintosh application and the A/ROSE Prep driver is made through the object routines, or glue code, in the A/ROSE Prep **library**. These routines provide for opening and closing the message queue to the driver, getting and freeing message buffers, and sending and receiving messages.

In addition, the A/ROSE Prep library provides access to many of the same utilities as provided by A/ROSE, such as moving data, obtaining the operating environment, and registering and looking up task names through the A/ROSE Prep Name Manager. These routines are located in the file `A/ROSE Prep:IPCGlue.o` on the MCP distribution disks. (All of these routines use the C calling sequence.)

A/ROSE Prep managers

The managers for A/ROSE Prep are the Echo Manager and the Name Manager. These A/ROSE Prep managers perform functions identical to and have the same message interface as their A/ROSE counterparts; minor differences are due to the slightly different interface to A/ROSE Prep.

The A/ROSE Prep managers are processes that carry out higher-level services on behalf of applications on the Macintosh computer. These managers are often referred to as **slot 0 managers**, and the Macintosh main logic board itself is sometimes referred to as the **slot 0 card.**

◆ *Note:* The slot 0 card is not to be confused with the Slot Manager in the Macintosh (part of the Macintosh Operating System).

Functions of MCP software

The functions of MCP software include the following:

- using messages for interprocess communication
- using the client/server relationship as a mechanism for data transfer
- using task scheduling in the A/ROSE multitasking environment
- managing memory under A/ROSE

Using messages for interprocess communication

Messages are the fundamental means for communication between A/ROSE tasks and A/ROSE Prep processes. Message structures are allocated from and returned to a special area of memory dedicated to holding messages. Intracard messaging is accomplished through the operating-system kernel; intercard messaging is handled by ICCM and A/ROSE Prep.

Message structures

A **message** is a fixed-length data structure that is sent between tasks. Some of the fields in a message indude

- a destination address, which is the identifier of the task to which the message is directed
- a source address, which is the identifier of the task that sent the message
- a message code specified by the task that sent the message
- three long words of data for the task to which the message is directed
- three long words of data that should be returned untouched in a response to the task that sent the message
- a pointer to a data buffer
- the size of the data buffer
- a message identifier
- message priority
- message status

Some of the fields in a message structure in C are:

```
long                 mId;          /* Message ID */
short                mCode;        /* Message code */
short                mStatus;      /* Message return status */
unsigned short       mPriority;    /* Message priority */
tid_type             mFrom;        /* Message source */
tid_type             mTo;          /* Message destination */
unsigned long        mSData[3];    /* Sender’s private data */
unsigned long        mOData[3];    /* Sender's shared data */
long                 mDataSize;    /* Size of data buffer */
                                   /* in bytes */
char                 *mDataPtr;    /* Address of data */
```

Figure 3-3 illustrates the fields contained in fixed-length messages for A/ROSE and A/ROSE Prep.

■ **Figure 3-3** Fixed-length message structure

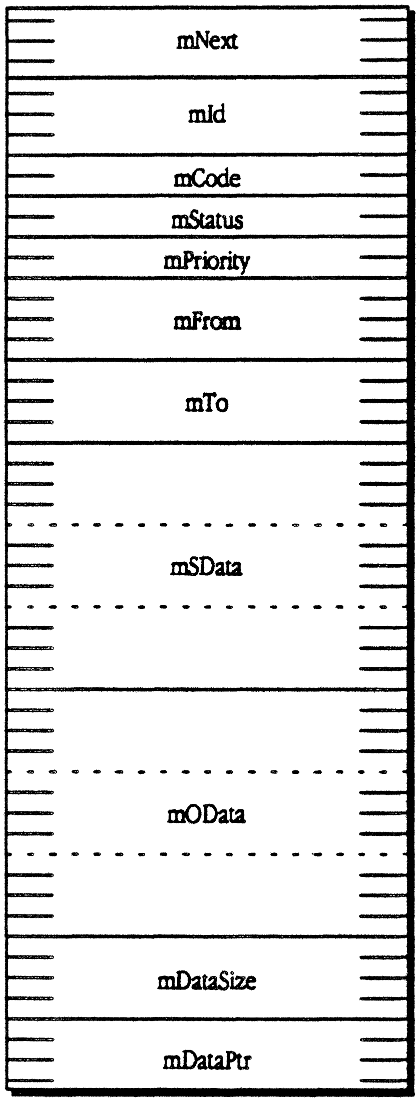


Table 3-1 describes some of the fields in the message structure and provides a brief description of each.

◆ *Note:* Always use the message structure as defined in the `includes` file.

■ **Table 3-1** Structure for fixed-length messages

Field Name	**Field Size**	**Description/Usage**
`mNext`	`Ptr`	a pointer used internally by A/ROSE for linking message buffers that are in a queue. While the message buffer is being used by the application, the mNext field can serve any function.
`mId`	long	a statistically-unique, 32-bit number to identify the message, initialized when a message is obtained from

A/ROSE or the A/ROSE Prep driver by way of a `GetMsg()` request. Your applications should never modify the message ID field of a message

Field Name	Field Size	Description/Usage
mCode	short	a 16-bit message code understood only by the sender and receiver of a message By convention, an even mCode is a request message, and an odd mCode is a reply message. You can find examples of this convention in the files :A/ROSE:includes:managers.a and :A/ROSE:includes:managers.h. For example, the ICCM request code ICC_GETCARDS(150) is even; the ICCM reply code ICC_GETCARDS+1(151) is odd. The Name Manager request code NM_REG_TASK(100) is even; the Name Manager reply code NM_REG_TASK+1 is odd. The A/ROSE operating system, the A/ROSE Prep driver, and the managers (Name Manager, ICCM, and others) set the high bit of the mCode in a message if the mCode is not recognized or the message is undeliverable. User tasks should also set the high bit if the message code was not recognized. The file managers.a and the file managers.h in the folder :A/ROSE: includes: list the mCodes known by A/ROSE, the A/ROSE Prep driver, and the managers.
mStatus	short	a 16-bit status code, with the upper 8 bits of mStatus designated as an A/ROSE system status code and the lower 8 bits of mStatus designated as a user status code. The mStatus values used by A/ROSE, A/ROSE Prep, and the managers are found in the files managers.a and managers.h in the folder :A/ROSE:includes:. User tasks should set the mstatus to OS_Unknown_Message if the message code was not recognized. For any message that is undeliverable, A/ROSE and A/ROSE Prep change the entire mStatus word to a value of $8000. If a message with mStatus already set to $8000 is found to be undeliverable, A/ROSE and A/ROSE Prep free the message.
mPriority	short unsigned	a 16-bit unsigned word representing the priority of the message (0 is the lowest priority)
mFrom	long	a source address (the task that sent the message) By convention, mFrom is the Task Identifier (TID) of the task sending the message. A/ROSE automatically fills in the mFrom field to that of the current TID when a message is obtained by a GetMsg() request. A task receiving a

message should swap the `mFrom` and `mTo` fields before sending a message in reply.

Field Name	Field Size	Description/Usage
		To declare the TID number, use `tid_type TYPEDEF` described later in this chapter. Do not assume anything about the format of fields in the TID. For example, the slot number may not always appear in the same location of the TID.
`mTo`	long	a destination address (the task to which the message is directed) The `mTo` field is the Task Identifier (TID) of the task to which you want to send a message. This field *must* be filled in before doing a `Send` request. To declare the TID number, use `tid_type TYPEDEF` . Do not assume anything about the format of fields in the TID. For example, the slot number may not always appear in the same location of the TID.
`mSData`	3 long words	12 bytes of data defined by the sender, associated with the message, that should be returned unchanged and unexamined by the receiver in a reply message. This field contains internal context information meaningful only to the tasks that sent the request. By convention within A/ROSE, a task receiving a request message copies the three `mSData` words from the request to the `mSData` words of the reply message. The task receiving the request should *not* otherwise manipulate this `mSData`.
`mOData`	3 long words	12 bytes of data defined by the receiver, associated with the message. By convention, these 3 long words are meant to be used between the requesting task and the replying task for passing information.
`mDataSize`	long	the size of an associated data buffer pointed to by `mDataPtr`. This size is in 8-bit bytes.
`mDataPtr`	long	a pointer to an associated data buffer.

Messages are obtained by a `Receive` request in the following order:

1. The message must fit any match criteria that was specified in the `Receive` request.
2. The highest `mPriority` message fitting the match criteria is obtained.

◆ *Note:* If two or more messages fitting the match criteria have the highest `mPriority`, the first one received and queued for the task is obtained (as in a First-In/First-Out, or FIFO, queue).

Mechanisms for data transfer

Data is transferred between tasks by one of three mechanisms: in the message code, in three long words in the message, or in a data buffer. A task may use all three mechanisms simultaneously when sending a message. Here is a description of these three mechanisms:

- **the message code**

  Through bilateral agreement between cooperating processes, the message code alone may convey the entire meaning of the message.

- **three long words in the message**

  The second mechanism allows a task to pass three long words of data in the message (`mOData[0]`, `mOData[1]`, and `mOData[2]`) whose meaning is specified by the receiving task (refer to the Timer Manager on the MCP distribution disk for an example).

  In addition, the task may pass another three long words of data in the message (`mSData[0]`, `mSData[1]`, and `mSData[2]`) that the receiver returns untouched. The `mSData` long words are private to the sending task; these words are not altered by the receiving task and should be returned to the requesting task unchanged. This feature allows tasks to pass context and other information, such as return addresses for processing, for the sending task's private use within the messaging mechanism.

  ◆ *Note:* This passing of information by tasks for private use is a convention; it is not enforced by the A/ROSE operating system.

- **a data buffer**

  The third mechanism involves passing the address of a data buffer and its size (that is, its length in bytes) in the message to the receiving task for it to use. The address of the buffer is placed in `mDataPtr` and the size of the buffer is placed in `mDataSize`.

  In an environment that includes intercard communications, `mDataPtr` could be pointing to an off-card buffer. The MCP card supports 32-bit accesses; however, with some other smart cards, all reads and writes to off-card buffers from a 32-bit CPU must be made with accesses of 16 or fewer bits.

Message and status codes

Table 3-2 lists message and status codes, with a brief description.

■ **Table 3-2** Message and status codes

Field	Size	Description/Comments
mCode	16-bit	message code field □ the upper bit is reserved for undeliverable messages □ use an even number to request services □ use an odd number for replies
mStatus	16-bit	message status field □ the upper 8 bits are used for passing operating-system status □ use the lower 8 bits used passing user status

The reply `mCode` to a request for service is the original `mCode`, plus 1.

The `Receive` system call uses message code 0 to indicate a match of any value. Therefore, you should not use message code 0 in the `mCode` field, as the field cannot be explicitly matched. By convention, the message code 0xFFFF (–1) is not used.

When a message cannot be delivered, the operating system changes the message code and message status as follows:

- the message code bit 1<<15 is set (`mCode` | 0x8000)
- the message status is assigned a value of 0x8000

If the operating system is unable to return the message to the sender (that is, if the sender has stopped or does not exist), the operating system frees the message but not any buffer associated with the message (pointed to by `mDataPtr`).

A task that receives a message it does not recognize must check if (`mCode` & 0x8000) is true (bit 1 << 15 is set).

- If true, the message was undeliverable and should be released via `FreeMsg()`. Any buffer associated with the message must not be released. This requirement ensures that messages will not loop and shared buffers are not freed.
- If false, `mCode` should be modified by setting bit 1<< 15 (`mCode` | 0x8000). The message status, `mStatus`, should be set to `OS_UNKNOWN_MESSAGE`. The task should then swap the source and destination TIDs and return the message to the sender.

The client/server relationship

The life of a typical message buffer begins in the message buffer pool. This message buffer pool is available to any task that may request a message buffer from the system.

When a task sends a message, it either utilizes a message buffer it owns (usually the message buffer it just received) or requests a message buffer from the system using a `GetMsg()` call. After filling the message with required addressing information and data, the task sends the message to its destination with a `Send` system call. The sending task has then lost rights to the message buffer, and it should not read from or write into the message buffer (or otherwise use the message buffer).

Upon receipt, the destination task either reutilizes the message buffer for an outgoing message, or returns it to the message buffer pool using a `FreeMsg()` call.

Client and server running on a smart card under A/ROSE

This section provides an example of a client and server running on a smart card under A/ROSE. You can find the source code for this example in the folder :A/ROSE: Examples:. The client is a timing test found in the `timeit.c` file; the server is the Echo Manager (similar to the echo example found in the `echo.c` file). (See the file `MakeFile` in the folder :A/ROSE:Examples: for making the `echo.c` and `timeit.c` examples.)

Both tasks are started within `osmain`, the main program, during A/ROSE initialization. The server first uses the A/ROSE utility `Register_Task` to register its name so that clients can find it. The server then enters its main loop and issues a `Receive` utility, waiting for messages from clients.

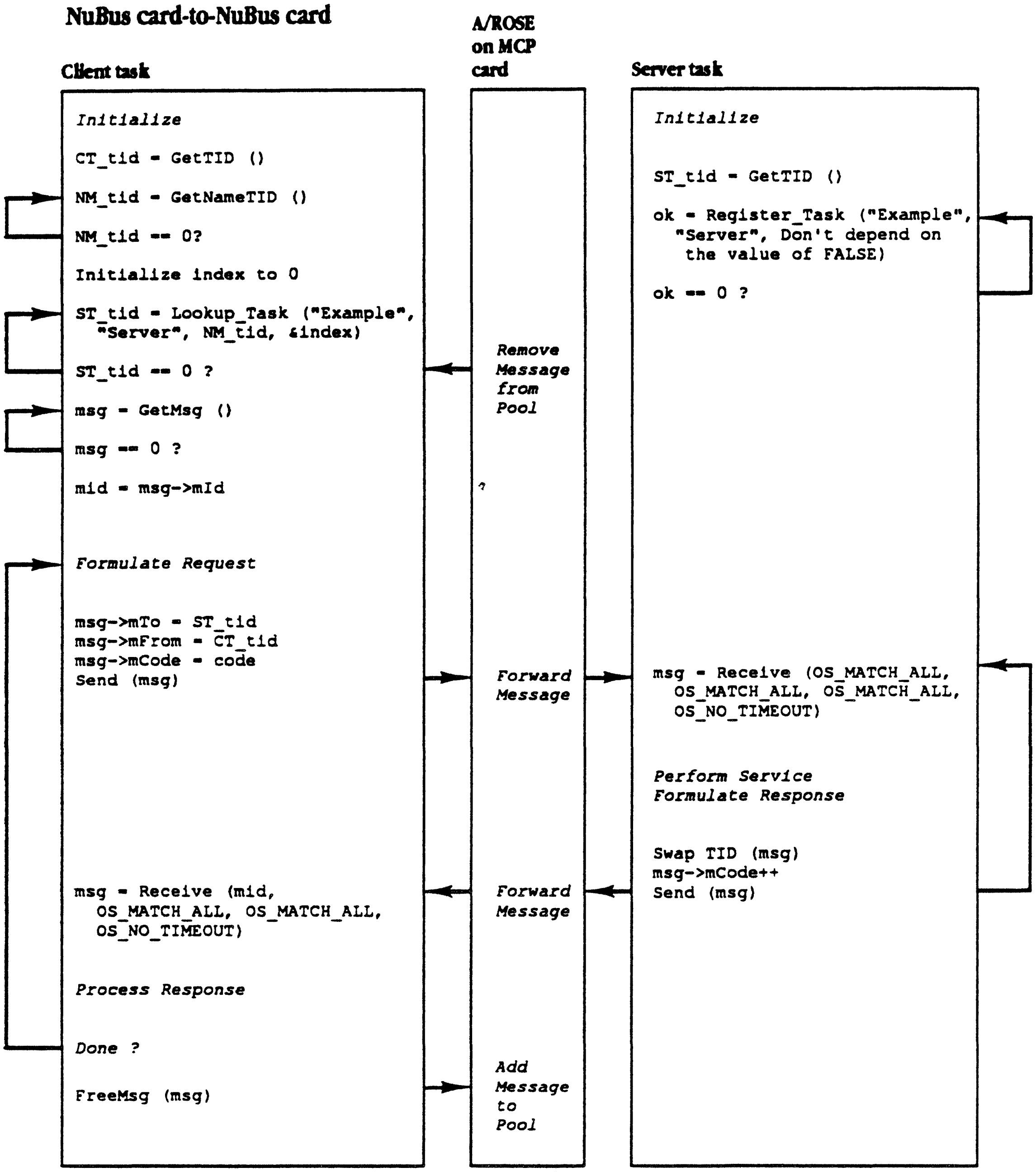
A client locates the server it wants to communicate with, using A/ROSE `Lookup_Task` utility to obtain the TID of the server. The client next obtains a message buffer, stores the TID of the server into the `mTo` field of the message buffer, sets the desired `mCode` request in the message buffer, and uses the `Send` utility to send the message buffer to the server. Next, the client issues a `Receive` to wait for a reply from the server.

The server receives the message, takes any action that is required of it, swaps the contents of the `mFrom` and `mTo` fields of the message, sets an appropriate `mCode` reply in the message buffer, and uses the `Send` utility to send the message buffer to the client. The server next issues a `Receive` utility to wait for another message from a client.

The client receives the reply from the server and takes appropriate action.

Figure 3-4 illustrates the client/server relationshp for tasks running under A/ROSE on the MCP card.

■ **Figure 3-4** Client/server relationship for A/ROSE program modules



NuBus card-to-NuBus card

Client task:

```
Initialize

CT_tid = GetTID ()

NM_tid = GetNameTID ()

NM_tid == 0?

Initialize index to 0

ST_tid = Lookup_Task ("Example",
  "Server", NM_tid, &index)

ST_tid == 0 ?

msg = GetMsg ()

msg == 0 ?

mid = msg->mId


Formulate Request


msg->mTo = ST_tid
msg->mFrom = CT_tid
msg->mCode = code
Send (msg)


msg = Receive (mid,
  OS_MATCH_ALL, OS_MATCH_ALL,
  OS_NO_TIMEOUT)


Process Response


Done ?

FreeMsg (msg)
```

Server task:

```
Initialize


ST_tid = GetTID ()

ok = Register_Task ("Example",
  "Server", Don't depend on
   the value of FALSE)

ok == 0 ?


msg = Receive (OS_MATCH_ALL,
  OS_MATCH_ALL, OS_MATCH_ALL,
  OS_NO_TIMEOUT)


Perform Service
Formulate Response


Swap TID (msg)
msg->mCode++
Send (msg)
```

Client and server running on Macintosh using A/ROSE Prep

The sequence of actions needed for a client and server running on the Macintosh using A/ROSE Prep is similar to that described earlier. This section also describes some of the differences between an application program running on the Macintosh and program modules running under A/ROSE on the MCP card.

Server and client processes using the A/ROSE Prep driver on the Macintosh are different from server and client processes running under A/ROSE because of the differences between A/ROSE and the Macintosh operating system; that is, A/ROSE is a multitasking operating system, and the Macintosh operating system assumes that there is a single application.

The source code for the example discussed in this section is found in the file `MakeFile` in folder `:A/ROSE:`A/ROSE Prep`:Examples`, as follows:

- For the client, source code for a timing example can be found in the `timeit.c` file (Timeit is an MPW tool)
- For the server, source code for'the Echo Manager can be found in the `os.o` file

The Echo Manager is started during INIT31 resource processing.

A server or client running under A/ROSE automatically has a TID associated with it; a server or client using the A/ROSE Prep driver on the Macintosh must first make itself known to the driver by issuing an `OpenQueue()` request. The `OpenQueue()` request makes the task known to the driver and assigns the requesting task a TID. The server in this example registers its name with the Name Manager as it did under A/ROSE so that clients can find it.

Under A/ROSE, both the server and the client can issue a blocking `Receive` request. A/ROSE has separate stacks for each task and saves each task's registers when switching between tasks. Using A/ROSE Prep on the Macintosh, only one process at a time (either the server or the client) can issue a blocking `Receive` request. Since the Macintosh operating system assumes that there is a single application, it will not switch to another application while one application is waiting for something to finish.

Using the A/ROSE Prep driver on the Macintosh, the `Receive` calling sequence includes an extra parameter. This parameter is the address of a completion routine to be called when the A/ROSE Prep driver receives a message that satisfies the `Receive` request. A task not using a completion routine to receive messages and not blocking must periodically issue a nonblocking `Receive` request to determine if there are any messages for it.

In the case of A/ROSE Prep Echo Manager code, the server issues a `Receive` request with a completion routine specified. The code following the `Receive` request exits the server; effectively, the server is no longer running. The server becomes a dangling piece of code tucked away in memory, called by the A/ROSE Prep driver when the driver receives a message satisfying its `Receive` request.

◆ *Note:* The `echo.c` file has no A5 references within it. An assembly language routine is used to access `echo.c` globals.

The client locates the server it wants to communicate with, using `Lookup_Task` to obtain the TID of the server. The client next obtains a message buffer, sets the TID of the server into the `mTo` field of the message buffer, sets the desired `mCode` request in the message buffer, and uses the `Send` request to send the message to the server. The client then issues a `Receive` request to wait for a reply from the server, specifying the address of a completion routine.

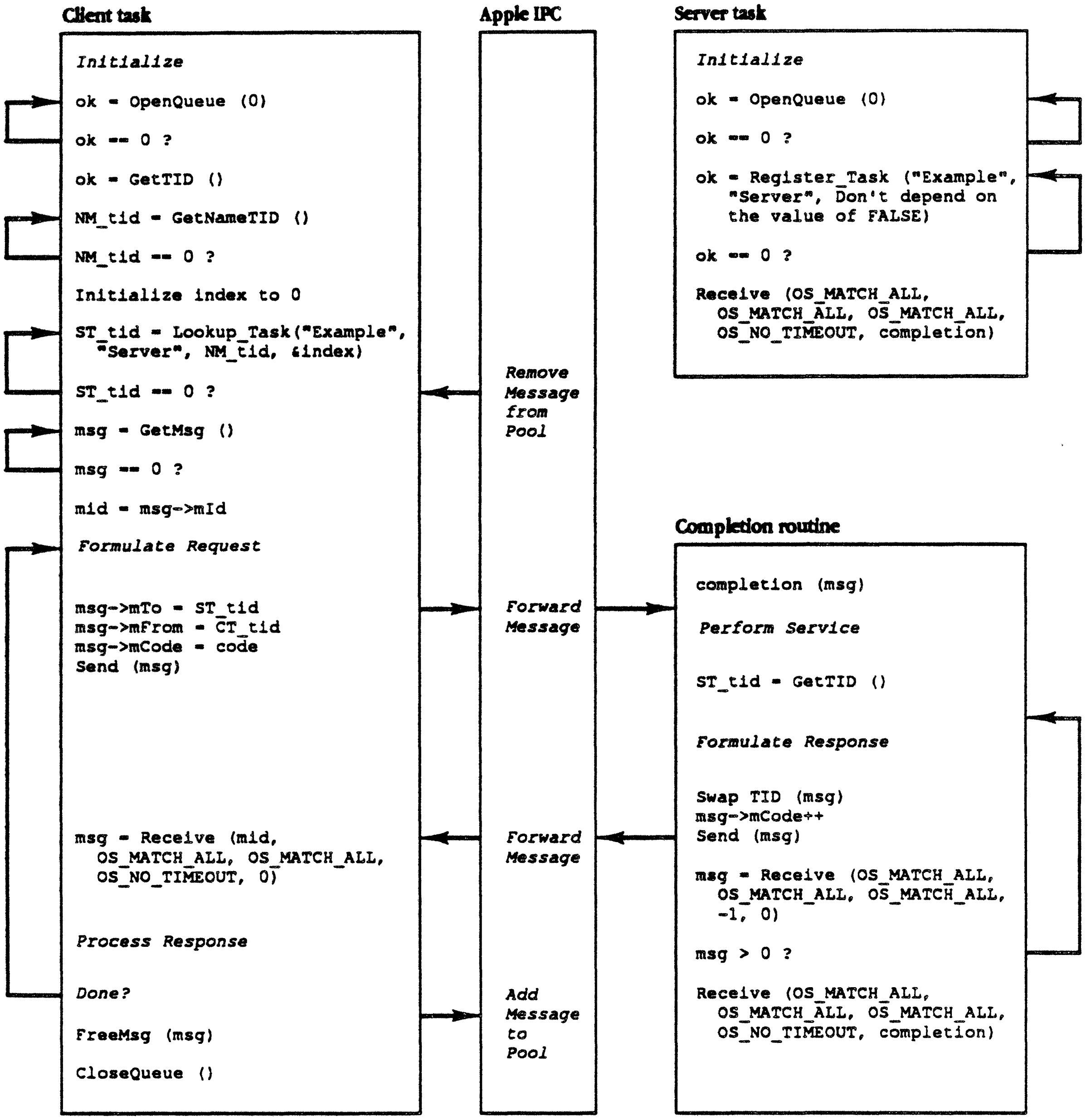
The A/ROSE Prep driver calls the server at the server's completion routine address, passing the message to the server. The server takes any action required of it, swaps the contents of the `mFrom` and `mTo` fields of the message, sets an appropriate `mCode` reply in the message buffer, and uses the `Send` request to send the message buffer to the client. The server must be carefully designed in how it handles the completion routine, since the completion routine may be called from an interrupt.

The client receives the reply from the server and takes appropriate action. The client then issues a `CloseQueue` request to notify the A/ROSE Prep driver that the client is finished talking to the A/ROSE Prep driver.

Figure 3-5 illustrates the client/server relationship for applications using the A/ROSE Prep driver. The first `Receive` request in the completion routine processes all messages in the queue. When there are no more messages, the second `Receive` request specifies the same completion routine again so that the routine will be called when there is another message.

◆ *Note:* Two `Receive` requests are specified so that the stack will not be overrun.

■ **Figure 3-5** Client/server relationship for applications using the A/ROSE Prep driver
Macintosh to Macintosh



```
Initialize

ok = OpenQueue (0)

ok == 0 ?

ok = GetTID ()

NM_tid = GetNameTID ()

NM_tid == 0 ?

Initialize index to 0

ST_tid = Lookup_Task("Example",
  "Server", NM_tid, &index)

ST_tid == 0 ?

msg = GetMsg ()

msg == 0 ?

mid = msg->mId

Formulate Request


msg->mTo = ST_tid
msg->mFrom = CT_tid
msg->mCode = code
Send (msg)


msg = Receive (mid,
  OS_MATCH_ALL, OS_MATCH_ALL,
  OS_NO_TIMEOUT, 0)


Process Response


Done?

FreeMsg (msg)

CloseQueue ()
```

```
Initialize

ok = OpenQueue (0)

ok == 0 ?

ok = Register_Task ("Example",
   "Server", Don't depend on
   the value of FALSE)

ok == 0 ?

Receive (OS_MATCH_ALL,
  OS_MATCH_ALL, OS_MATCH_ALL,
  OS_NO_TIMEOUT, completion)
```

```
completion (msg)

Perform Service

ST_tid = GetTID ()

Formulate Response

Swap TID (msg)
msg->mCode++
Send (msg)

msg = Receive (OS_MATCH_ALL,
  OS_MATCH_ALL, OS_MATCH_ALL,
  -1, 0)

msg > 0 ?

Receive (OS_MATCH_ALL,
  OS_MATCH_ALL, OS_MATCH_ALL,
  OS_NO_TIMEOUT, completion)
```

Using task scheduling in a multitasking environment

A **task** is a message-driven transaction processor that runs on the MCP card. The behavior of a task depends on the messages it receives.

Tasks include the Idle Task; managers, such as the ICCM, Name Manager, Print Manager, Remote System Manager, Timer Manager, and Trace Manager; and any developer-written tasks.

Task Identifiers

Tasks are known by and referred to A/ROSE by Task Identifiers (TIDs). These identifiers are for internal use and are automatically assigned by A/ROSE when it starts a task.

Modes in which tasks run

There are two modes in which tasks run:

- tun-to-block mode (also referred to as **block mode**)
- **slice mode**

In run-to-block mode, a task has control of the CPU until the task explicitly releases it, either by changing its scheduling parameters (using a `Reschedule` call), or by waiting to receive a message (using a blocking `Receive` call) or by using an A/ROSE library routine that waits for a response to a message (`printf`, `Lookup_Task`, and so forth). The purpose of run-to-block mode is to guarantee uninterrupted use of the CPU to tasks that need it; an example of a place where you should use run-to-block mode is in critical sections of code.

◆ *Note:* Do not confuse run-to-block mode with the blocking receive operation in which a message is awaited. The name "run-to-block" captures the idea that the task holds onto the processor until it performs a blocking receive. A ***blocked task*** is one that waits for a message, having performed a blocking `Receive`.

In slice mode, the task can be *time-sliced*; that is, the operating system temporarily suspends execution of the task to allow tasks of equal or higher priority to run.

A task can change its running mode as necessary by using the A/ROSE primitive `Reschedule()`, see Chapter 4.

Timer services

You can schedule tasks using timer services provided by A/ROSE. For timer services and message reception done with a timeout, time is specified in major **ticks.** A major tick is the smallest time unit recognized by tasks in the operating system. This value is specified in all blocking `Receive` and timing operations.

▲ **Caution** All code segments that have been installed in the Tick chain run when a major clock tick is detected by the operating system. These segments are executed even if the current task is in run-to-block mode. Refer to Chapter 7 for more information about the Tick chain. ▲

Task scheduling

Tasks are scheduled in round-robin fashion in each priority ring. There are 32 priorities, ranging from 0 (lowest) to 31 (highest). The operating system scans the priority table, beginning at the highest priority, for a task that is eligible to run. Tasks with the same priority are scheduled on a first-come, first-served basis. Over time, this scheduling allows all tasks in a priority ring to be given an equal opportunity to execute. Tasks of equal priority therefore share the processor.

A task of higher priority can indefinitely keep a lower priority task from executing, but in common practice, a task always does a blocking `Receive` that permits lower priority tasks to execute. Obviously, priorities of tasks must be chosen carefully, so that the most critical tasks have the highest priorities. A task may change its scheduling mode by using a `Reschedule` call.

Scheduling decisions are made at every major tick of the system clock.

- If the current task is in slice mode, it can be preempted; that is, another task with a higher priority can take precedence over the task running in slice mode. If a high-priority task is available (not blocked), that task will be scheduled before the lower-priority task running in slice mode.
- If the current task is in run-to-block mode, it is always allowed to continue.

Task initialization

During initialization, a task performs whatever functions may be necessary for its execution. Every task has different needs, but typical functions include

- setting its scheduling mode as necessary
- waiting for other required tasks to begin
- registering its name with the Name Manager

The choice of scheduling mode depends on the function the task performs:

- Slice mode is used for tasks that are pre-emptible. Time-slicing of such tasks permits other tasks to share the CPU.
- Run-to-block mode is for tasks that, because of time constraints or the need to be protected during critical sections of code, cannot give up the CPU to other tasks.

◆ *Note:* Tasks can take exclusive control of the CPU only in situations where other tasks do not need to execute; if other tasks are ever to execute, the task must change its scheduling mode or perform a blocking receive to free the CPU.

In response to its needs, a task can change its scheduling mode as it executes.

A/ROSE always creates one task during its initialization; that task is the **Idle task**. The Idle task increments a counter, calls the Idle Chain, and issues the `Reschedule` primitive to allow tasks to run. The Idle task runs in block mode, and is given the lowest priority (priority 0). When no other task is eligible for execution on the processor, A/ROSE schedules the Idle task. Code segments can be run when A/ROSE is idle by installing them in the Idle Chain (refer to Chapter 7 for more information).

▲ **Caution** The Idle task must always be eligible for execution. The system halts if it can find no tasks to schedule; hence a `StopTask` should not be performed on the Idle task. ▲

Task execution

The bulk of a task is a message loop in which a message is waited for, received, and processed. Actually, a message is both waited for and received through the `Receive` primitive.

Task termination

If a task must terminate, it notifies the operating system via a `StopTask` call. `StartTask` initializes a task such that, if the main routine returns, a `StopTask` is automatically issued.

Memory managment

To increase performance of the A/ROSE operating system, developers must make a distinction between using general-purpose memory and using message buffers.

For general-purpose memory, the available pool of memory extends from the last address in which the operating-system code is loaded, up to the system stack area that occupies the last `cOSStack` bytes of RAM. The system stack occupies the last portion of RAM and the stack is the size you specify. Therefore, the amount of memory available in the pool depends on the size of the code and data spaces. You can allocate and free general-purpose memory to tasks using the `GetMem` and `FreeMem` calls, described in Chapter 4.

During initialization, the operating system sets aside a block of memory large enough to hold the maximum number of messages that you specified. This block of memory is then linked together to form the free list of messages. Messages can be quickly allocated and released from this list. You specify the number of messages allocated to the operating system in the call to `osinit()` from `main()`.

When using this document to manage memory on MCP, be aware of the following specialized terminology used in this document. The terminology refers to locations on the main logic board of Macintosh and the MCP card. The main logic board in any Macintosh computer is called *slot 0.* In the descriptions that follow, *slot* refers to the Macintosh main logic board or any smart NuBus card. This document also uses the term *NuBus address* even though some members of the Macintosh family do not actually have a NuBus.

Background on virtual addressing with A/ROSE

This section shows how virtual addressing functions on Macintosh, particularly with A/ROSE. Each smart NuBus card has a 680x0 processor and local memory; a virtual address is the address used by a task to reference its own memory.

For example, assume a task is running on a smart NuBus card that has a 68000 processor. The card is in slot 0x0d. The task has a buffer in local memory at address 0x00005000 and can reference this local memory using address 0x00005000. Since a 68000 processor ignores the most significant byte of every address, the task can also reference this local memory using virtual address 0xfd005000 or any virtual address 0xMN005000, where M and N are any hex digits. Each of these addresses (0x00005000, 0xfd005000, and 0xMN005000) is a virtual address that the task can use to refer to its own memory. In the example just mentioned, the task has multiple virtual addresses for the same memory.

A NuBus address is the address of the memory location through the nubus.

◆ *Note:* A card can be constructed so that not all local memory addresses can be accessed from the NuBus. A card can also be constructed so that certain card addresses can be accessed from the NuBus but cannot be accessed locally.

In this example (which could be called a flat address space), assume a task is running on a smart NuBus card in slot 0x0d and the task has a buffer in its local memory at address 0x00005000. Two possibilities exist for the NuBus address (the actual NuBus address depends on the hardware on the NuBus card in slot 0x0d).

One address could be 0xfd005000. This is the Minor NuBus address of the buffer. The buffer is in slot 0x0d. Local addresses on a card that respond to Minor NuBus addresses *all* have 0xfd as the most significant byte of the address.

The other address could be 0xd0005000. This is the Major NuBus address of the buffer. Local addresses on a card that respond to Major NuBus addresses *all* have 0xdM as the most significant byte of the address.

NuBus cards, however, can respond to *both* Minor and Major NuBus addresses.

◆ *Note:* The Apple MCP card only responds to Minor NuBus addresses.

The descriptions of the A/ROSE Prep services and the A/ROSE primitives and utilities refer to several different types of memory addresses. The types of virtual addresses used in the remainder of this document are described in the next four sections. Only NuBus addresses and 32-bit virtual addresses (with associated TID) can be passed freely between slots in a Macintosh computer.

Flat address space

A flat address space means that the *virtual* address of any byte in memory is the same as the *real* address of the byte.

In the previous example, a flat address space would be the instance where the task in slot 0x0d referenced its local buffer using address 0xfd005000 and the *real* address of the buffer was 0xfd005000.

◆ *Note:* The MCP card uses a flat address space for local addresses.

Some Macintosh computers come with 68030 processors containing a PMMU. A PMMU allows the implementation of some special memory management functions, described in the following two examples.

◆ *Note:* On a Macintosh computer containing a PMMU, NuBus accesses from smart cards to memory on the Macintosh main logic board to Macintosh main board memory *do not* go through the PMMU.

In this example, assume a task is running on the Macintosh main logic board containing a PMMU. Assume the task has a local buffer at its virtual address 0x00005000. Because a PMMU exists, the buffer *does not* have to be at real address 0x00005000. The buffer could be at real address 0x00060000, for example. In fact, a buffer does not even have to be contiguous in real memory. Assume that the buffer has a very large capacity. The large buffer can still start at virtual address 0x00005000. Assume the large buffer goes through virtual address 0x0000F000. In this case, virtual address 0x00005000 through 0x00007FFF could be at real addresses 0x00060000 through 0x00067FFF. In addition, Virtual address 0x00008000 through virtual address 0x0000F000 could be at real address 0x00080000 through 0x00087000.

At the lowest level, NuBus cards accessing memory on the Macintosh main board must use NuBus addresses and access real memory.

Notice how the use of virtual memory is possible using a PMMU and a backing store such as a disk drive. Assume a task is running on the Macintosh main board and the task has a local buffer at virtual address 0x00005000. With a PMMU and backing store, the buffer *does not* have to be in real memory unless the buffer is being accessed by the task. The buffer can also be at different *real* memory locations each time the buffer is paged in from disk.

24-bit virtual addresses

A **24-bit virtual address** identifies a location within the address space of a Macintosh main logic board. All versions of the Macintosh operating system use 24-bit virtual addresses. Applications running on the Macintosh main logic board use 24-bit virtual addresses internally when accessing C and assembly language variables. <What about applications that don't run on the main logic board?>>

On the Macintosh main logic board, a 24-bit virtual address itakes the form "m0aa aaaa", where "m" contains bits used by the Memory Manager. Since the Memory Manager modifies these bits independently of the active application, all 24-bit virtual addresses whose lower 24 bits are identical refer to the same memory location.

24-bit virtual addresses cannot be passed to smart NuBus cards. An application running on the Macintosh main logic board must call the Macintosh operating system trap `_StripAddress` to convert a 24-bit virtual address to a 32-bit virtual address before passing the address to a NuBus smart card.

▲ **Caution** A/ROSE does not support the 24-bit "NuBus-like" addresses used by applications such as MacsBug. These 24-bit addresses are of the form "00sa aaaa", where "s" is the number of the slot containing the memory location.. ▲

32-bit virtual addresses

A **32-bit virtual address** identifies the **local address space** of a particular slot. A 32-bit virtual address takes the form "aaaa aaaa", where all bits are treated as part of the address.

Because the RAM on the main logic board of a Macintosh IIci is not contiguous, consecutive 32-bit virtual addresses can refer to memory locations with very different NuBus addresses.

When the 32-bit virtual address is passed to a task on another slot, the TID of that task (*on the slot containing the memory location*) must also be passed to provide contextual information. Because of the unusual NuBus address space of the Macintosh IIci, 32-bit virtual addresses are recommended for passing addresses between slots.

The A/ROSE, MapNuBus, LockRealArea, UnlockRealArea, and NetCopy primitives and services discussed later in this document all require 32-bit addresses. These A/ROSE primitives and utilities take 32-bit virtual addresses as input parameters and return 32-bit virtual address results.

▲ **Caution** *Do not* use 24-bit addresses with A/ROSE functions. The Macintosh operating system trap `_StripAddress` should be used on the Macintosh main logic board to convert 24-bit addresses, which may contain memory manager information, into 32-bit addresses. ▲

NuBus address

A NuBus address uniquely identifies the entire **NuBus address space** of the Macintosh computer. The Macintosh recognizes a distinction between two types of NuBus addresses: minor NuBus addresses and major NuBus addresses. The **minor NuBus address** takes the form "Fsaa aaaa", where "s" is the number of the slot containing the location. The **major NuBus address** takes the form "saaa aaaa", where "s" is the number of the slot containing the location.

The main logic board of a Macintosh computer responds only to NuBus accesses using major NuBus addresses. MCP-based smart NuBus cards respond only to NuBus accesses using minor NuBus addresses. However, Smart NuBus cards can be designed to respond to either minor or major NuBus addresses, or both.

The NuBus address of a memory location can be passed freely between slots without additional contextual information.

RAM on the main logic board on all Macintosh computers, except the Macintosh IIci, is contiguous: the main logic board of an 8MB Macintosh IIx, for example, contains RAM at major NuBus addresses 0x00000000 through 0x007FFFFF. A Macintosh with contiguous RAM is said to have a **flat address space** (discussed earlier in this section).The main logic board of the Macintosh IIci contains discontiguous RAM. As a result, the RAM on Macintosh IIci does not have a flat address space.

▲ **Caution** A NuBus address cannot *arbitrarily* be used to access more than one byte of memory on the Macintosh IIci main logic board. To ensure compatibility with the Macintosh IIci, your application should avoid using NuBus addresses wherever possible.. ▲

Latched virtual address

The **latched virtual address** is unique to MCP-based cards and other smart NuBus cards that cannot directly access the entire NuBus address space of a Macintosh computer (see Chapter 12, *MCP Card Specifications*). A latched virtual address is of the form "00Aa aaaa". The upper 12 bits of a latched virtual address are the actual hexadecimal digits 0x00A.

Despite the "virtual" in its name, a latched virtual address suffers from the same disadvantages as a NuBus address on the Macintosh IIci <<what are the disadvantages?>>. In addition, latched virtual addresses are subject to memory boundary restrictions that make them even more awkward and dangerous to use.

Before accessing an off-card (off the MCP card) memory directly from code running on an MCP-based card, make sure that some task calls the A/ROSE MapNuBus utility. In addition, make sure MapNuBus contains the NuBus address of the memory location you're trying to access. The MapNuBus utility sets the MCP-based card's page latch registers for the task and returns a latched virtual address. The latched virtual address is used in accessing the off-card memory location. The latched virtual address is valid only for the task that called the MapNuBus utility and only until the task's next call to the MapNuBus utility. A/ROSE saves and restores the card's page latch registers during task scheduling.

▲ **Caution** On MCP-based NuBus cards, the A/ROSE MapNuBus utility sets up a one-megabyte window *beginning on a one-megabyte boundary*. If you must use latched virtual addresses, your task must issue a new call to MapNuBus when the low 20 bits of the latched virtual address overflow..

In addition, you cannot *arbitrarily* use a latched virtual address to access more than one byte of memory on the Macintosh IIci main logic board. To ensure compatibility with the Macintosh IIci, your application should avoid using latched virtual addresses whenever possible. ▲

Virtual memory support

The physical memory of the Macintosh IIci is noncontiguous so a memory management unit is used to map a contiguous virtual memory onto noncontiguous physical memory. However, only accesses by the main Macintosh logic board processor are mapped, leaving smart card processors with the problem of being able to deal only with physical memory.

Solutions to the Macintosh IIci support problem can conveniently be applied when working with virtual memory in System 6.0.4.

In System 6.0.4.., the Macintosh operating system provides a call to acquire contiguous and, in the case of virtual memory, frozen and locked memory. Locked memory means that the memory area will always be in the physical memory and will not be swapped out to disk; however, the memory area can move within the real memory. "Frozen" memory means the memory area will stay in the same place and will not be eligible for paging to the backing store. Also, the operating system will provide a way to unlock and unfreeze this memory through another call. A/ROSE provides these system calls to provide access to main memory from smart cards.

Extreme care should be exercised by users of the A/ROSE `MapNuBus` function. Direct access of Macintosh main logic board memory from the card is hazardous because to use NuBus addresses means that you must lock pages in physical memory for the period that the NuBus address is to be used. As more pages are locked in physical memory, fewer are available for swapping. This can result in an overall reduction in performance if too many pages become ineligible for swapping since those pages are locked. As a limit, if all pages are locked, the system will crash on the next page fault, because no more pages are eligible to be swapped out. When you use `BlockMove` to move data quickly, (the only situation where you might use `MapNuBus)`, you must be aware that the memory addresses you use might be virtual addresses and might not correspond to the actual NuBus addresses. Two new routines are provided to help users run A/ROSE in the Macintosh IIci and virtual memory environments. These routines are `LockRealArea()`, and `UnlockRealArea()`, (both described in Chapter 4). You must use `LockRealArea()` and `UnlockRealArea()` calls to get the actual NuBus addresses. The `BlockMove` routine is a simple assembly language loop that moves long words from one area to another and so is very fast. Developers must take care to set up the addresses for the call so that the addresses are real and are present.

In the Macintosh IIci and virtual memory environments, the `CopyNuBus()` utility would not be safe and has been removed from A/ROSE. The `NetCopy()` utility (described in Chapter 5) copies data from one virtual address to another safely, but requires two additional parameters to specify the context of the virtual source and destination addresses. These two parameters are the TaskIDs of the source and destination tasks.

◆ *Note:* The virtual address passed to the `A/ROSE Prep`, `LockRealArea` services and `UnlockRealArea` must be 32-bit clean addresses and must not contain memory manager bits. In addition, the virtual address must be in the same address space as the process that calls these services.

The A/ROSE Prep driver has been augmented to provide calls corresponding to the Macintosh operating system calls that acquire and free contiguous memory. By having applications go through A/ROSE Prep to acquire and free their buffers, A/ROSE Prep can access the information needed for optimization.

The existing `MapNuBus()` routine within A/ROSE will not be changed in the Macintosh operating system running in virtual memory environment.

The input parameter to `MapNuBus()` continues to be a NuBus address. `MapNuBus()` will return an address that must be used to reference this NuBus address.

Chapter 4 **A/ROSE Primitives**

THIS CHAPTER describes the operating-system primitives used for A/ROSE. A primitive is similar to a system call, in that a primitive provides fundamental services from the operating system. Primitives are invoked as hardware traps and thus operate in supervisor, or server, mode. ■

Table 4-1 lists the primitives provided by A/ROSE and gives a brief description of each.

■ Table 4-1 A/ROSE primitives

Name	**Description**
`FreeMem()`	Frees a block of memory
`FreeMsg()`	Frees a message buffer
`GetMem()`	Allocates a block of memory
`GetMsg()`	Allocates a message buffer
`Receive()`	Receives a message
`Reschedule()`	Changes a task's scheduling mode
`Send()`	Sends a message
`Spl()`	Sets the hardware-priority level
`StartTask()`	Initiates a task
`StopTask()`	Stops a task

These primitives are calls that are made by most tasks running under A/ROSE. Some primitives are used in `main()`, the program that executes before anything else starts. You can modify `main()` for whatever application you are writing.

■ Table 4-2 **Macintosh Operating System Calls**

Macintosh Operating System Calls	**Function**
`_StripAddress`	The `_StripAddress` call converts a 24-bit virtual address to a 32-bit virtual address. A 32-bit virtual address passed to `_StripAddress` is returned unchanged.
`_SwapMMUMode`	The `_SwapMMUMode` call toggles the addressing mode of the Macintosh main logic board between 24-bit virtual and 32-bit virtual addressing.

◆ *Note:* A/ROSE uses C calling conventions, and all registers are preserved except `D0`, `D1`, `A0`, and `A1`. The assembly-language macros also adhere to these conventions.

Operating system primitives

This section describes each of the operating system primitives and provides examples of how to call primitives from both C and assembler. Both calling sequences take arguments and use similar data structures.

FreeMem( )

`FreeMem()` frees a block of memory that was acquired earlier by a call to `GetMem()`. A/ROSE decrements the usage count associated with the buffer. If the resulting usage count is zero, the memory is returned to the free memory pool; if the usage count is non-zero, the memory is not released.

The C declaration of `FreeMem()` is

```
void            FreeMem( ptr )
char            *ptr;                /* pointer to memory buffer to free */
```

The form for the FreeMem macro is as follows, where `P1` is the address of the memory block to be freed:

```
[Label]         FreeMem         P1
```

P1 can be specified as a register (`A0-A6`, `D0-D7`), or can use any 68000 addressing mode valid in an LEA instruction to specify the location containing the desired address.

▲ **Caution** A/ROSE will execute an illegal instruction if an attempt is made to free a memory buffer that has not been allocated by A/ROSE. ▲

FreeMem will return to the caller doing nothing if PTR is NIL <<What does this mean, doing nothing?>>

FreeMsg( )

FreeMsg() frees a message buffer that was acquired earlier by a call to GetMsg().

The operating system distinguishes between messages and memory in order to speed up the acquisition and disposal of messages. The number of messages initially available depends upon the number requested in the call to osinit() from main().

The C declaration of FreeMsg() is

```
void           FreeMsg( mptr )
message        *mptr;               /* pointer to message buffer to free */
```

The form for the FreeMsg macro is as follows, where P1 is the address of the message buffer to be freed:

```
[Label]        FreeMsg              P1
```

P1 can be specified as a register (A0-A6, D0-D7), or can use any 68000 addressing mode valid in an LEA instruction to specify the location containing the desired address.

▲ **Caution** In most cases, A/ROSE will execute an illegal instruction if an attempt is made to free a message after it has been sent and when a message buffer that has not been allocated by A/ROSE is freed using FreeMsg( ). ▲

GetMem( )

`GetMem()` requests a block of memory from the free memory pool. The size of the free memory pool depends upon the size of the program or code space loaded and the amount of memory installed on the card.

`GetMem()` returns either a pointer to the allocated block of memory or 0. If the memory could not be allocated, a call to `FreeMem()` releases the memory. The allocated message block is initialized to 0 by `GetMem()` if the number of bytes requested is greater than 0; otherwise, the memory is not initialized. For example, `GetMem(-10)` returns a pointer to a block of 10 bytes. `GetMem(10)` returns a pointer to a block of 10 bytes that have been initialized to zero. The usage count associated with the buffer is set to 1. (See A/ROSE utilities `GetUCount` and `IncUCount` in Chapter 5.)

The C declaration of `GetMem()` is

```
char        *GetMem( size )
long        size;              /* size of block to allocate */
```

The form for the GetMem macro is as follows, where `P1` is the size of the memory block to be allocated:

```
[Label]     GetMem                  P1
```

P1 can be specified as a register (`A0-A6, D0-D7`), or an immediate value (`#<abs expr>`), or can use any 68000 addressing mode valid in an LEA instruction to specify the location of a long word holding the desired block size. The address of the allocated block is returned in `D0` unless the block could not be allocated, in which case 0 is returned in `D0`.

GetMsg( )

`GetMsg()` requests a message buffer from the free message pool. `GetMsg()` either returns a pointer to the allocated message or 0 if the message could not be allocated. A call to `FreeMsg()` releases the message.

A/ROSE clears all fields in the message, except `Message ID` (`mID`) and `From` address (`mFrom`), before the pointer to the message is returned. `Message ID` (`mID`) is set to a number that is statistically unique to the field. `mFrom` is set to the current task identifier.

The C declaration of `GetMsg()` is

```
        message         *GetMsg()
```

The form for the `GetMsg` macro is

```
        [Label]         GetMsg
```

The address of the allocated message buffer is returned in `D0` unless no buffer was available. In that case, `0` is returned in `D0`.

LockRealArea( )

LockRealArea enables the Netcopy call to use information for performance improvements

△ **Caution** LockRealArea must be called by the task in the same virtual address space as the buffer to be locked and frozen in memory. △

Once called, LockRealArea creates a table entry informing A/ROSE, running on NuBus smart cards, of any areas that are successfully locked. LockRealArea then returns the *physical* addresses and lengths of the memory areas that are successfully locked in memory. <<how does this benefit programmers who want to use MapNubus?>>

LockRealArea should be used to lock a memory buffer that needs to be accessed frequently and quickly. LockRealArea is especially useful because it speeds up NetCopy (described in chapter 5) requests. In the future, you might be required to use LockRealArea to lock down memory buffers residing on smart NuBus cards.

LockRealArea should be called when the buffer is obtained to lock the memory in place. LockRealArea is a slow operation. Trial and error will determine when best not to use LockRealArea. LockRealArea will lock memory so it cannot be paged. Do *not* use LockRealArea to lock down an infrequently-used buffer that does not have to be accessed quickly. Do *not* use LockRealArea to lock down large buffers because of the dangers of page-fault thrashing.

`LockRealArea()` will accept as input a virtual address and a length and will attempt to lock the memory associated with this virtual address into physical memory. The virtual address must be in the same address space as the routine that called `LockRealArea`. Therefore, a NuBus card cannot lock memory on the Macintosh main logic board. Likewise, another NuBus card and the mother board cannot lock memory on any other NuBus card. If successful, the physical address/length pairs of the memory associated with the virtual address are returned.

The structure addressareas, in which LockRealArea returns the physical address/length pair is defined as the following:

```
struct  addressareas

{

        void *address;    /* Physical address of memory area */
        unsigned long length;      /* Length of memory area */

};
```

The calling sequence for LockRealArea() is the following:

```
short LockRealArea(void *virtualaddr,unsigned long length,
             struct addressareas *buffer, unsigned long count);
```

virtualaddr is the virtual address of the memory area to be mapped. length is the length of the memory area. buffer is the area where the physical address map is returned. buffer is a pointer to an array of structure addressareas. count is the number of physical address/length pairs (addressareas structures) that the buffer can hold. This is the same as the number of elements in the array addressareas.

You can declare the buffer in the following way:

```
struct addressareas buffer[16];
```

If the size of the buffer[] is large enough for only one entry; that is, count has a value of one, the pages are forced to be contiguous. Otherwise, the pages may not to be contiguous when locked in memory.

The physical address/length pairs are returned in buffer. Any unused address/length entries in the buffer are initialized to zero.

LockRealArea() returns zero if successful. If the pages could not be locked and frozen in physical memory or if the buffer was not large enough to contain the entire physical address map then LockRealArea() will either return an error code of "erLockFailed" or an error code returned by Macintosh Operating System. The memory is neither locked nor frozen if an error is returned.

UnlockRealArea( )

UnlockRealArea is the inverse operation of LockRealArea. Only buffers that were previously locked using LockRealArea can be unlocked using UnlockRealArea.

UnlockRealArea should be called when the buffer is no longer going to be shared across the NuBus. UnlockRealArea is a slow operation.

▲ **Caution** UnlockRealArea must be called by the task that locked the buffer in memory. ▲

UnLockRealArea takes a virtual address and a length as parameters, and will attempt to unlock and unfreeze the memory associated with this virtual address into physical memory. If successful, the physical address(es) of the pages associated with the virtual address are returned.

The structure address area is defined in the following way:

`UnlockRealArea()` unlocks a memory area that was previously locked with a call to `LockRealArea()`.

The calling sequence for `UnlockRealArea()` is

```
short_ UnlockRealArea(void *virtualaddress, unsigned long length);
```

`virtualaddress` is the beginning virtual address of an area of memory that was previously locked and frozen. `length` is the length of the memory area that was locked and frozen.

`UnlockRealArea()` returns zero if the address was successfully unlocked. Otherwise, an error is returned.

◆ *Note:* The address and length parameters specified in a call to `UnlockRealArea()` must exactly match the virtual address and length specified in a previous call to `LockRealArea()`. `UnlockRealArea()` cannot handle fragmented unlocking in this release, that is, you cannot unlock a portion of a previously locked and frozen memory area.

`UnlockRealArea()` returns a status of zero if the virtual memory support is not available.▲

Receive( )

`Receive()` returns the highest priority message from the task's message queue that matches the specified criteria. Like the `Reschedule` primitive, `Receive` may be used to enable the CPU to run other tasks. Unlike `Reschedule`, `Receive` allows tasks of lower priority to run.

The C declaration of `Receive()` is

```
message*Receive( mID, mFrom, mCode, timeout )
unsigned long mID;          /*    Unique message ID to wait on      */
tid_type mFrom;             /*    Sender address to wait on         */
unsigned short mCode;       /*    Message code to wait on           */
long timeout;                     /*    Time to wait in major ticks
*/                                /*    before giving up
*/
```

The first three criteria (`mID`, `mFrom`, and `mCode`) may be set to match either a specific value (by specifying the value), or to match any value (by specifying the symbol `OS_MATCH_ALL`), or to no value (by specifying the symbol `OS_MATCH_NONE`).

The timeout parameter in major ticks takes one of the three values described here:

- A value of timeout < 0 requests a nonblocking `Receive`. A nonblocking `Receive` returns control immediately to the task, regardless of whether a message matching the criteria was found or not. If no message was found, ø is returned. Any negative value can be used.
- A value of timeout = 0 requests a blocking `Receive` with no timeout. This `Receive` returns control only when a message matching the criteria is found.
- A value of timeout > 0 requests a blocking `Receive` with a timeout. This `Receive` returns when either the timeout parameter expires or a message matching the criteria is received, whichever occurs first. A timeout returns ø.

The form for the `Receive` macro is as follows, where `P1` is the message ID match code, `P2` is the sender address match code, `P3` is the message code match code, and `P4` is the timeout code:

```
[Label]        Receive        P1, P2, P3, P4
```

P1 through P4 can each be specified as a register (`A0-A6`, `D0-D7`) or an immediate (`#<abs-expr>`) or it can use any 68000 addressing mode valid in an LEA instruction to specify the location of a long word containing the desired value. The address of the returned message buffer is returned in `D0` unless no message was available. In that case, `0` is returned in `D0`.

The following example shows how to use the `Receive` primitive in your code segment to delay a task for five seconds:

```
        Receive (OS_MATCH_NONE, OS_MATCH_NONE, OS_MATCH_NONE,
                        5*GetTickPS());
```

The `Receive` criteria for message ID, sender's address, and message code must never be satisfied in order to delay for a specified period of time. After every five seconds, A/ROSE causes the task to be eligible for execution. To implement a delay, you can use a `Receive` with matching criteria that can match no message.

△ **Important** Take care using the `mCode` selector in `Receive` requests. The operating system will set bit 15 of mCode (mCode | 0x8000) when a message cannot be delivered. If a task does a `Receive` and waits on `mCode`, `Receive` will never see its message criteria matched if the message is undeliverable; hence the program will never get what it's waiting for. It's better to wait on `message ID (mID)`, because the operating system does not change this field. △

Reschedule( )

The `Reschedule()` primitive is used to give tasks of the same or higher priority a chance to run before scheduling the task that issues the `Reschedule` call. `Reschedule()` never causes tasks of lower priority to run.

`Reschedule()` selects the operating mode of the task, which can be any one of the options listed in *Table 4-3*. Block mode differs from slice mode only in that the task will not give up the CPU until the task is explicitly blocked by `Receive()` or executes another call to `Reschedule()`.

- **Table 4-3** Reschedule options

Option	New scheduling mode	Schedule a higher-or equal priority task before returning to the task that issued the Reschedule request?
`OS_SLICE_MODE`	Slice	Yes
`OS_BLOCK_MODE`	Block	Yes
`OS_SLICE_IMMED`	Slice	No
`OS_BLOCK_IMMED`	Block	No
`OS_RTN_MODE`	Does not change	Yes
`OS_RTN_IMMED`	Does not change	No

`OS_SLICE_MODE` changes the scheduling mode of the task to time-slice scheduling, and allows any higher-priority or equal-priority task to execute before this task executes again.

`OS_BLOCK_MODE` changes the scheduling mode of the task to run-to-block scheduling mode, and allows any higher-priority or equal-priority task to execute before this task executes again.

`OS_SLICE_IMMED` changes the scheduling mode of the task to time-slice scheduling mode, and continues execution of this task until the next time-slice interval, when normal task scheduling occurs.

`OS_BLOCK_IMMED` changes the scheduling mode of the task to run-to-block mode, and continues execution of this task until the task blocks itself by doing another `Reschedule` or a blocking `Receive` request.

`OS_RTN_MODE` returns the current scheduling mode of the task without changing the scheduling mode, and allows any higher-priority or equal-priority task to execute before this task executes again.

`OS_RTN_IMMED` returns the current scheduling mode of the task, and continues execution of the current task without attempting to schedule any other higher-priority or equal-priority task.

The C declaration of `Reschedule()` is

```
short       Reschedule( mode )
short       mode;           /* Scheduling mode */
```

Reschedule returns the previous scheduling mode.

The form for the `Reschedule` macro is as follows, where `P1` specifies the new operating mode of the task:

```
[Label]        Resched        P1
```

P1 can be specified as a register (`A0-A6`, `D0-D7`), an immediate value (`#<abs-expr>`), or use any 68000 addressing mode valid in an LEA instruction to specify the location of a long word containing the desired operating mode. The previous scheduling mode is returned in `D0`.

`Reschedule` may be useful when combined with a nonblocking `Receive` request to give other tasks a chance to run, as shown in the following example.

This example describes how to use `Reschedule` for two tasks implementing two different layers of the X.25 protocol. Suppose one task implements X.25 Level 2; the other task implements X.25 Level 3. In this example, both tasks execute with the same scheduling priority. The Level 2 task is operating in block scheduling mode; the Level 3 task is operating in either time slice or block scheduling mode and should not depend on what the Level 2 layer is doing.

Accordingly, a portion of the Level 2 task might look like the following:

```
(
message       *msg   *m;

/* Initialize and send message to level 3 task indicating data is present

*/

If (msg=GET MSG ())

(

        /* Fill in msg to send */

                    ...

        Send (msg);

        m = Receive(OS_MATCH_ALL, OS_MATCH_ALL, OS_MATCH_ALL, -1);
                                   /* See if data present from Level 3 */
        Send(m);                   /* Send data to Level 3 task */

        if (m == 0)                /* If nothing from Level 3 yet */
        (
            Reschedule(OS_BLOCK_MODE); /* Let Level 3 task execute */
            m = Receive(OS_MATCH_ALL, OS_MATCH_ALL, OS_MATCH_ALL, -1);
                                /* Try to get data from Level 3 */
        }

        /* Three cases exist:
         *1. No information was available; m = 0
         *2. Information was previously available from Level 3 before we
         *    did the Send; m = address of message
         *3. Level 3 task had enough time to provide information after
         *    we did the send; m = address of message
         */
        /* See if data present from level 3 */
        if (m != 0)
        {
            /* If Level 3 task has information to be sent, */
            /* send I frame message with information. */
        }
        else
        {
            /* If Level 3 did not have information to be sent, */
            /* send RR frame. */
        }
}
```

The Level 2 task gives up the CPU by way of the `Reschedule` request in order to allow the Level 3 task to execute. In the case of an X.25 implementation, this could allow Level 2 acknowledgements to be piggy-backed with data from Level 3.

Send( )

Send() places a message on the task's queue specified by the message field, mTo. The message is placed in the queue in priority order (from highest to lowest).

▲ **Caution** In most cases, A/ROSE executes an illegal instruction if an attempt is made to send a message that is not available to a task for sending. For example, do not send the same message twice; also, do not send a message and then free it. ▲

The C declaration of Send() is

```
void          Send( mptr )
message       *mptr;              /* pointer to message buffer */
```

If a message is undeliverable, it will be returned to the sender with the message status (mStatus) set to 0x8000 and the message code (mCode) having bit 15 set.

◆ *Note:* Send() assumes that all fields have been filled in (mFrom, mTo, mCode, and so forth) when this call is made.

The form for the Send macro is as follows, where P1 is the address of the message buffer to be sent:

```
[Label]       Send                P1
```

P1 can be specified as a register (A0-A6, D0-D7) or use any 68000 addressing mode valid in an LEA instruction to specify the location containing the address of the message buffer to be sent.

Spl( )

Programmers modify the status register to temporarily disable interrupts; A/ROSE provides the Spl() system call to allow user-mode tasks to set the hardware interrupt-priority level.

Tasks are always executed in the 68000's user mode, while interrupt routines and main() are executed in supervisor mode. This process is important because some 68000 instructions cannot be executed in user mode (such as any instruction that explicitly modifies the status register).

While a task is running with an elevated (non-zero) interrupt priority, it temporarily behaves as if it is in run-to-block mode.

▲ **Warning** Depending upon the elevated priority, interrupt handlers may still execute. ▲

In addition, if the task calls Receive and blocks with an elevated priority level, the priority level of the hardware is changed to the priority level of the next task that A/ROSE schedules. Therefore, you should not call Receive with an elevated priority level.

Spl() expects an integer from 0 to 7, and returns the previous priority as an integer from 0 to 7 (0 is the lowest interrupt priority and 7 is the highest interrupt priority).

The C declaration of Spl() is

```
short          Spl( npr )
short          npr;                  /* New interrupt priority */
```

The form for the SIL macro is as follows, where P1 specifies the new interrupt priority (0 to 7):

```
[Label]        SIL P1;               not Spl
```

▲ **Caution** The name of the macro is SIL, not the 68000 instruction Spl to avoid any conflict with the 68000 instruction. ▲

P1 can be specified as a register (A0-A6, D0-D7), an immediate value (#<abs-expr>), or can use any 68000 addressing mode valid in an LEA instruction to specify the location of a long word containing the desired interrupt priority level. The previous interrupt priority level is returned in D0.

StartTask( )

`StartTask()` is used to create a task and make it eligible for execution. `StartTask()` returns either the task identifier of the created task, or 0 if the task could not be created. The new task is initially started in slice mode.

The C declaration of `StartTask()` is

```
      tid_type      StartTask( STpb)
      struct        ST_PB   *STpb;
```

The format of the parameter block referenced by `*STpb` is shown next.

```
struct ST_PB

{

      char      *CodeSegment;           /* memory region on card for code      */

      char      *DataSegment;           /* memory region on card for     */
                                        /* global data      */

      char      *StartParmSegment;      /* memory region on card for */
                                        /* start parameters */

      struct    ST_Registers InitRegs;  /* initial register set for      */
                                        /* starting task   */

      long      stack;                  /* initial stack size (in bytes) */

      long      heap;                   /* initial heap size (in bytes) */

      short     return_code;            /* error code if task not started */
                                        /* (Tid = 0)   */

      unsigned char priority;           /* priority of task */

      tid_type  ParentTID;              /* TID of Parent on Network/Host */

};

struct ST_Registers
{
      long D_Registers [8];      /* D0 - D7    */
      long A_Registers [8];      /* A0 - A7          Note: A7 not used */
      long PC;                   /* Program Counter */
}
```

These parameters include the following:

- `priority`, which is the scheduling priority at which the task will run. There is currently no way to change this priority once a task is created. Priority 0 is the lowest; priority 31 is the highest.
- `stack`, which is the size of the task's stack in bytes. There is no way to change this size after execution of `StartTask()`.
- `heap`, which is the amount of heap storage in bytes that the task will need to start up. Using `heap` prevents tasks from coming up and not being able to run due to lack of memory. The pointer to this storage is accessible via `GetHeap()`.
- `ParentTID`, the task ID of the task that is designated as the parent of the running task; use `GetTID()` to obtain the TID to be used for the parent TID.

The parameter block contains pointers to up to three memory segments that must have been previously allocated by calls to `GetMem()`.

In all cases, `CodeSegment` and `DataSegment` must be zero if the task being started was linked into the operating system.

If the task was not linked into the operating system, you must issue a `GetMem()` or an `RSMGetMem()` request to reserve the space for the code segment. The `CodeSegment` parameter must be set to the value returned by `GetMem()`. If the task was linked to the operating system, set the `CodeSegment` parameter to zero.

A `GetMem` request must be issued to reserve space for the `DataSegment`, if the `DataSegment` is present. The `DataSegment` must be set to the value returned by `GetMem()`, or zero if the `DataSegment` is not present.

If there are parameters, a `GetMem` request must be issued to get memory for the `StartParmSegment`. `StartParmSegment` is set to zero if there are no start parameters to pass to the task; otherwise, the `StartParmSegment` must be set to the value returned by `GetMem()`.

The registers hold the initial values of the registers when the task is started. The value specified for Register A7 is not used; the value is replaced by the pointer to the stack when the task is started. The program counter contains the absolute address of the start code.

The task is initially started in slice mode. If the task was not started (if it returns 0), the return code specifies the reason, as shown here:

```
STE_NO_ERRORS              /* The start task functions */
                     /* successfully      */

STE_NO_TCB           /* No room in task table or */
                     /* no memory available for stack */
                     /* or heap */
```

▲ **Warning** `FreeMem()` must not be called by your application to release the memory allocated for `CodeSegment`, `DataSegment`, or `StartParmSegment`, because releasing memory is done automatically by `StopTask()`. Refer to the section later in this chapter on `StopTask()` for more information. ▲

The form for the `StartTask` macro is as follows, where `P1` is the address of a `StartTask` parameter block:

```
        [Label]        StartTask          P1
```

`P1` can be specified as a register (`A0-A6`, `D0-D7`), an immediate (`#<abs-expr>`), or use any 68000 addressing mode valid in an LEA instruction to specify the location of a long word containing the address of the parameter block. The task ID of the started task is returned in `D0` unless the task could not be started, in which case 0 is returned in `D0`.

To start a task on a different smart card that is also running A/ROSE, send a message to the Remote System Manager on the other card to reserve memory for the task; download the task to the card; then send messages to the Remote System Manager to start executing the task.

StopTask( )

StopTask() kills a currently executing task. StopTask() is automatically called to kill the task when the task fails or returns from the task's main().

If the task was started with any CodeSegment, DataSegment, or StartParmSegment, StopTask() calls FreeMem() to release each memory buffer.

The C declaration of StopTask() is

```
void            StopTask( tid )
tid_type        tid;                /* Task ID to kill */
```

The form for the StopTask macro is as follows, where P1 specifies the task ID of the task to stop:

```
[Label]         StopTask            P1
```

P1 can be specified as a register (A0-A6, D0-D7) or as an immediate value (#<abs-expr>) or it can use any 68000 addressing mode valid in an LEA instruction to specify the location of a long word containing the desired task ID.

The task identifier specified must not be that of the idle task (TID = 0), and it must be a task running on the requester's card.

If a task calls StopTask() and specifies its own task identifier, the task will cease functioning and stop your program. To stop a task on a different smart card that is also running A/ROSE, send a message to the Remote System Manager on the other card.

▲ **Warning** If one task stops another task, that task being stopped will not have the opportunity to release any message buffers that it is currently processing.

Chapter 5 A/ROSE Utilities

This chapter describes the operating system utilities available with A/ROSE. A utility is a library code segment linked with your application. ■

Table 5-1 lists the A/ROSE utilities, and provides a brief description of each.

■ **Table 5-1** A/ROSE utilities

Name	Description
`BlockMove()`	Copies a block of data from the source physical address to the destination physical address
`ARoseDate2Secs()`	Calculates and returns the number of seconds given a specific date and time
`GetBSize()`	Returns the size of a memory buffer in bytes
`GetCard()`	Returns the NuBus slot number of the card on which the calling process or task is running
`AROSEGetDateTime()`	Returns the number of seconds between 12:00 P.M. (Midnight), January 1, 1904, and the time that the function was called
`GetETick()`	Returns the number of major ticks since the A/ROSE was started
`GetgCommon()`	Returns the address of the `gCommon` operating system data area
`GetHeap()`	Returns the address of the heap area allocated to the task
`GetICCTID()`	Returns the task identifier of the InterCard Communication Manager
`GetNameTID()`	Returns the task identifier of the Name Manager
`GetStParms()`	Returns the address of the calling task's Start Parameters
`GetTCB()`	Returns the address of the calling task's Task Control Block
`GetTickPS()`	Returns the number of major ticks in 1 second
`GetTID()`	Returns the task identifier of the calling task
`GetTimerTID()`	Returns the task identifier of the Timer Manager
`GetTraceTID()`	Returns the task identifier of the Trace Manager
`GetUCount()`	Returns the usage count associated with a buffer
`IncUCount()`	Increments the usage count associated with a buffer
`IsLocal()`	Returns an indication of whether or not an address is local
`Lookup_Task()`	Returns the task identifier of the task that matches the Object Name and the Type Name specified
`MapNuBus()`	Translates a NuBus address into a local address and sets any address-mapping hardware
NetcopyO	Takes virtual addresses for its address arguments
`Register_Task()`	Registers a task with the Object Name and the Type Name specified
`ARoseSecs2Date()`	Calculates and returns the corresponding date and time record, given a number of seconds
`SwapTID()`	Swaps the `mFrom` and `mTo` fields in a message buffer
`ToNuBus()`	Translates a local address into a NuBus address
`TraceReg()`	Registers the current task as the Trace Manager

A description of utilities

This chapter describes each of the operating system utilities and provides examples of the C declarations for each utility. This chapter also describes the assembler macros; these macros have a one-to-one relationship to the calls and require the same number of parameters. A/ROSE uses C calling conventions, and all registers are preserved except `D0`, `D1`, `A0`, and `A1`. A/ROSE macros adhere to this convention.

◆ *Note:* The routines `MapNuBus` and `ToNuBus` are hardware dependent. Code written in C that uses these calls may not be portable. Code written in Assembler that makes calls to `MapNuBus` and `ToNuBus` will *not* be portable.

Three date- and time-related routines are provided with A/ROSE; the calling sequences and structures for these routines are defined in the file `os.h` in the folder :A/ROSE:includes:. These routines are identical to the routines `AROSEGetDateTime()`, `AROSEDate2Secs()`, and `AROSESecs2Date()` within the Macintosh II operating system.

BlockMove( )

`BlockMove()` does a simple move of bytes from the source to the destination, without checking for overlapping source and destination addresses. The number of bytes is specified in `count`.

▲ **Caution** Overlapping the source and destination blocks could cause partial overwriting of the destination block. ▲

The C declaration for `BlockMove()` is

```
void BlockMove ( source, destination, count )

       char               *source;
       char               *destination;
       long               count;
```

The following example shows how to call `BlockMove` in assembly language.

```
       MOVE.L    #Count,-(A7)
       PEA       Destination
       PEA       Source
       JSR       BlockMove
       ADD.L     #12,A7
```

AROSEDate2Secs( )

AROSEDate2Secs() takes the given date/time record, converts it to the corresponding number of seconds elapsed since 12:00 P.M. (Midnight), January 1, 1904, and returns the result in the location whose address is contained in the secs parameter.

The C declaration for AROSEDate2Secs() is

```
    pascal void AROSEDate2Secs(Date, secs)
       AROSEDateTimeRec    *Date;
       unsigned long       *secs;
       extern;
```

The following example program shows how to use all three date/time utilities.

```
#include      "os.h"
main()
{
        unsigned long secs;
        AROSEDateTimeRec    dtrec;
        unsigned long newsecs;

        GetDateTime(&secs);
        Secs2Date(secs, &dtrec);
        Date2Secs(dtrec, &newsecs);

        printf(" Date = %d/%d/%d, Time = %d:%d:%d\n",
               dtrec.year, dtrec.month, dtrec.day,
               dtrec.hour, dtrec.minute, dtrec.second);

        printf("Secs = %d,  Day of week = %d, New secs = %d\n",
               secs, dtrec.dayOfWeek, newsecs);
}
```

The following example shows how to call Date2Secs in assembly language:

```
        PEA    Date          ; Address of Date/time record
        PEA    secs          ; Address for result
        JSR    Date2Secs
```

▲ **Caution** In the previous version of the operating system, a routine Date2Secs() was included to give code running on NuBus cards the same functionality as the Macintosh toolbox Date2Secs() call. Unfortunately, the parameters were declared different from the Macintosh toolbox call.

The A/ROSE calling sequence caused C to push the entire DateTimeRec structure onto the stack instead of pushing a pointer to the DateTimeRec structure. The code within the previous version of the operating system would then get the DateTimeRec structure off the stack.

To be compatible with MPW 3.0 C, A/ROSE passes a pointer to the DateTimeRec. The code that processes the Date2Secs() request has been changed to expect a pointer. ▲

GetBSize( )

The input to `GetBSize()` is a pointer to a memory data buffer. The pointer was obtained by a call to `GetMem()`. The output from `GetBSize()` is either the size of the buffer in bytes or 0. Each buffer has an associated buffer header that is not included in the value returned by `GetBSize()`.

`GetBSize()` accepts 0 as input and returns 0 as output. `GetBSize()` does not check the input pointer for validity. The C declaration for `GetBSize()` is

```
unsigned long      GetBSize ( buffer )
extern unsignedchar *buffer;      /*pointer to buffer */
```

The following example shows how to call `GetBSize` in assembly language:

```
                        ; buffer pointer in A4
MOVE.L    A4,-(A7)      ; move buffer address onto stack
JSR       GetBSize      ; get the buffer size
ADD.L     #4,A7         ; pop the stack
TST.L     D0            ; D0 has the size
BEQ.S     XXX           ; bad buffer
```

◆ *Note:* If a pointer to the buffer is given to `GetBSize()` which was not obtained through the `GetMem()` call, the return results are not predictable.

GetCard( )

GetCard() returns the NuBus slot number of the card on which the calling task is running.

The C declaration for GetCard() is

```
        char GetCard ();
```

The following example shows how to call GetCard in assembly language:

```
        JSR     GetCard
```

Upon return, D0 contains the slot number. The slot number is kept in location gSlotNum in the gCommon data area.

AROSEGetDateTime( )

`AROSEGetDateTime()` returns the number of seconds between 12:00 P.M. (Midnight), January 1, 1904, and the time that the function was called.

The C declaration for `AROSEGetDateTime()` is

```
extern pascal void AROSEGetDateTime(secs)
        unsigned long        *secs;
        extern;
```

The following example shows how to call `AROSEGetDateTime` in assembly language:

```
        PEA         secs                ; Address for result
        JSR         AROSEGetDateTime
```

Refer to the utility `AROSEDate2Secs()` earlier in this chapter for an example program that shows how to use each date/time utility.

GetETick( )

`GetETick()` returns the number of major ticks—that is, the elapsed time in ticks—since the operating system started.

The C declaration for `GetETick()` is

```
externunsigned long GetETick();
```

The following example shows how to call `GetETick` in assembly language. and shows the location of the number of major ticks

```
JSR    GetETick
```

Upon return, `D0` contains the number of major ticks since the operating system started.

GetgCommon( )

`GetgCommon()` returns the address of the A/ROSE operating system data area, `gCommon`. Refer to the include files on the MCP distribution disks for the structure of `gCommon`.

The C declaration for `GetgCommon()` is

```
extern struct gCommon *GetgCommon();
```

The following example shows how to call `GetgCommon` in assembly language.

```
JSR       GetgCommon
MOVE.L    DO -> A0          /* A0 contains the beginning */
                            /* address of the gCommon data area*/
```

The `gCommon` address is contained in the constant `gCommon`.

GetHeap( )

`GetHeap()` returns the address of the heap area allocated to the task. If no heap area has been allocated, `GetHeap` returns 0. The heap size is specified in a parameter to the A/ROSE `StartTask` utility.

The C declaration for `GetHeap()` is

```
        char        *GetHeap();
```

The following example shows how to call `GetHeap` in assembly language:

```
        JSR         GetHeap             ; on return, D0 has pointer to heap
        TST.L       D0                  ; check if heap present
        BEQ.S       XXX                 ; jump if no heap
```

▲ **Caution** `FreeMem()` must not be called by your application to release the heap area allocated, as this process is done automatically by `StopTask()`. ▲

GetICCTID( )

GetICCTID() returns the task identifier of the InterCard Communication Manager. If there is no ICCM registered, GetICCTID returns 0. The C declaration for GetICCTID() is

```
extern tid_type            GetICCTID ();
```

The following example shows how to call GetICCTID in assembly language. The task identifier of the InterCard Communication Manager is kept in the location gIccTask in the gCommon data area.

```
JSR        GetICCTID
```

Upon return, D0 contains the task identifier of the ICCM.

GetNameTID( )

`GetNameTID()` returns the task identifier of the Name Manager. The C declaration for `GetNameTID()` is

```
        extern tid_type     GetNameTID ();
```

The following example shows how to call `GetNameTID` in assembly language. The task identifier in the Name Manager is kept in the location `gNameTask` in the `gCommon` data area.

```
        JSR        GetNameTID
```

Upon return, `D0` contains the task identifier of the Name Manager.

GetStParms( )

GetStParms() returns the address of the calling task's Start Parameters. If the calling task has no StartParameter, GetStParms returns 0. The C declaration for GetStParms() is

```
extern char      *GetStParms ();
```

The following example shows how to call GetStParms in assembly language:

```
JSR       GetStParms        ; on return, DO has pointer to
                            ; Start Parameters
TST.L     DO                ; check if Start Parameters present
BEQ.S     XXX               ; jump if no Start Parameters
```

▲ **Caution** Your application must not call FreeMem() to release the memory allocated for its start parameters; this process is done automatically by StopTask(). ▲

GetTCB( )

`GetTCB()` returns the address of the calling task's Task Control Block (TCB). The C include files contain information on the TCB structure. The C declaration for `GetTCB()` is

```
extern struct pTaskSave *GetTCB ();
```

The following example shows how to call `GetTCB` in assembly language. The address of the calling task's Task Control Block is kept in location `gCurrTask` in the `gCommon` data area

```
JSR       GetTCB

Move.L    Do, so          ;address of task control bolock
```

GetTickPS( )

`GetTickPS()` returns the number of major ticks in one second. The C declaration for `GetTickPS()` is

```
        extern unsigned short GetTickPS ();
```

The following example shows how to call `GetTickPS` in assembly language. The number of major ticks in 1 second is kept in the location `gTickPerSec` in the `gCommon` data area.

```
        JSR        GetTickPS
```

```
Upon return Do contains the number of major ticks in one second
```

▲ **Warning** Because of hardware limitations, the number of major ticks per second multiplied by the length of one tick may not equal one second. ▲

GetTID( )

`GetTID()` returns the task identifier of the calling task.

The C declaration for `GetTID()` is

```
        extern tid_type     GetTID ();
```

The following example shows how to call `GetTID` in assembly language. The task identifier of the calling task is kept in the location `gTID` in the `gCommon` data area.

```
        JSR        GetTID

Upon return, Do contains task identifier of calling task.
```

GetTimerTID( )

GetTimerTID() returns the task identifier of the Timer Manager. If there is no Timer Manager registered, GetTimer returns Ø.

The C declaration for GetTimerTID() is

```
        extern tid_type      GetTimerTID ();
```

The following example shows how to call GetTimerTID in assembly language. The task identifier of the Timer Manager is kept in the location gTimerTask in the gCommon data area.

```
        JSR        GetTimerTID
```

```
Upon return, Do contains task identifier of calling task.
```

GetTraceTID( )

`GetTraceTID()` returns the task identifier of the Trace Manager. If there is no Trace Manager registered, then `GetTraceTID` returns 0.

The C declaration for `GetTraceTID()` is

```
        extern  tid_type            GetTraceTID  ();
```

The following example shows how to call `GetTraceTID` in assembly language. The task identifier of the Trace Manager is kept in the location `gTraceTask` in the `gCommon` data area.

```
        JSR         GetTraceTID

Upon return, Do contains task identifier of calling task.
```

GetUCount( )

`GetUCount()` provides information when one task is sending information to many tasks; that is, when there are multiple tasks sharing a buffer. `GetUCount()` returns the usage count associated with the buffer. The buffer must have been allocated by a call to `GetMem()`. The usage count starts at 1 and is incremented by calling the A/ROSE `IncUCount` utility. A return value of 0 indicates that the pointer passed was 0.

The C declaration for `GetUCount()` is

```
extern unsigned char        GetUCount ( buffer )
char            *buffer;    /* pointer to buffer */
```

The following example shows how to call `GetUCount` from assembly language:

```
MOVE.L   A0,-(A7)     ; push buffer address
JSR      GetUCount    ; usage count is returned in D0
ADD.L    #4,A7        ; pop the stack
```

◆ *Note:* If a pointer to a buffer not obtained through the `GetMem()` call is given to `GetUCount()`, the return results are not predictable.

IncUCount( )

`IncUCount()` is useful where buffers are shared between different tasks and a mechanism is needed to ensure the orderly release of the buffers. `IncUCount()` increments a buffer's usage count and returns the incremented usage count (when it has a value of 2 or greater) of the buffer, or 0. A return value of 0 indicates that the pointer passed was 0 or that the usage count has not been incremented because an overflow of the usage count field would have resulted. The buffer must have been allocated with a call to `GetMem()`. The usage count is decremented when the buffer is freed using `FreeMem()`.

The C declaration for `IncUCount()` is

```
unsigned char   IncUCount ( buffer )
char            *buffer;      /* pointer to buffer */
```

The following example shows how to call `IncUCount` in assembly language:

```
MOVE.L   A4,-(A7)     ; push buffer address
JSR      IncUCount    ; usage count is returned in D0
ADD.L    #4,A7        ; pop the stack
```

◆ *Note:* If a pointer to a buffer not obtained through the `GetMem()` call is given to `IncUCount()`, the return results are not predictable.

IsLocal( )

IsLocal() returns a true or false indication of whether or not a NuBus address is local.

The C declaration for IsLocal() is

```
extern short    IsLocal (address)
char            *address;     /* address to test. */
```

IsLocal() returns true (non-zero) if the NuBus address passed is local. IsLocal() returns false (zero) if the address passed is a remote Nubus address.

The form for the IsLocal macro is as follows, where P1 is the address to examine:

```
[Label]         IsLocal       P1
```

P1 can be specified as a register (A0-A6, D0-D7) or an immediate (#<abs-expr>) or it can use any 68000 addressing mode valid in an LEA instruction to specify the location of a long word containing the desired value.

Lookup_Task( )

`Lookup_Task()` returns either the task identifier of the task that matches the Object Name and Type Name specified, or 0 if no matching task is found. The wildcard character = is allowed. Initially, the index must be set to 0; on subsequent calls, it should be left unchanged. `Lookup_Task()` modifies the variable index; this index allows `Lookup_Task()` to find any additional entries that may match the criteria in subsequent calls.

▲ **Caution** `Lookup_Task()` communicates with the Name Manager and issues a blocking `Receive`; therefore, the task gives up control of the CPU during this call.. Do not use the `Lookup_Task` utility in a completion routine or from a routine on the tick or idle chain. ▲

The C declaration for `Lookup_Task()` is

```
tid_type     Lookup_Task  (object, type, nm_TID, index)
char         object[];
char         type[];
tid_type     nm_TID;
unsigned     short *index;
```

The task identifier of a Name Manager is `nm_TID`, and may be obtained by using `GetNameTID()` or by sending the message `ICC_GetCards` to the ICCM. `Lookup_Task()` returns the task identifier of the first task that matches the criteria.

The following code provides an example of how to look up all tasks on the current card:

```
short index   ;
tid_type tid  ;

index = 0     ;
while ((tid = Lookup_Task ("=", "=", GetNameTID (), &index)) > 0)
        printf ("TID %x Found \015\012", tid);
```

The following example shows how to call `LookupTask` in assembly language:

```
MOVE.W     #0,INDEX    ; initialize index
PEA        INDEX       ; address of index
MOVE.L     TID,D0      ; value of tid on stack
MOVE.L     D0,-(A7)    ; place on stack
PEA        TYPE_NAME   ; address of type name
PEA        OBJECT_NAME; address of object name
JSR        Lookup_Task
ADD.W      #16,A7      ; pop the stack
TST.L      D0          ; check if found
BNE.S      D0,XXX      ; jump if found
```

MapNuBus( )

MapNuBus exists only under A/ROSE running on the smart NuBus card. MapNuBus does not exist within the A/ROSE Prep driver running on the Macintosh main board. The operating system preserves the state of the page latch registers if any, for each task.

The input parameter to MapNuBus is the NuBus address of a buffer that is guaranteed to be frozen and locked. MapNuBus sets up page latch registers, if any, and returns a 32-bit virtual address that can be used for accessing the physical memory location.

MapNuBus is dangerous to use. Use MapNuBus only when speed is most important. Most programmers can use NetCopy (described next) in place of MapNuBus with great satisfaction .Programmers choosing to use MapNuBus *must* know the hardware limitations of the smart NuBus card being used. For example, on certain smart NuBus cards, such as the MCP card, the page latch register moves a NuBus address space window. The NuBus address space window for the MCP card is one-megabyte long and always begins on a one-megabyte boundary. A new MapNuBus request must be done each time a one-megabyte boundary is crossed.

Programmers choosing to use MapNuBus *must* also know information about the buffer to be accessed. For example, the buffer on the Macintosh main board might be paged out to disk unless the programmer takes special care that it is not paged.

`MapNuBus()` translates a pointer that may contain a NuBus address to a local pointer. This local pointer is used by the calling task to access the associated data. `MapNuBus()` also sets up any address mapping hardware required for the access. The local pointer is only valid for the task that called `MapNuBus` because each task may set up the address mapping hardware differently.

- ◆ *Note:* The local pointer is hardware specific. See Part II for details on the numeric value or the bounds on the value.

`MapNuBus()` passes through 0 and local addresses without modifying them. You should assume that only a single off-card mapping for a task is active at any given time on each card; each call to `MapNuBus()` by a particular task invalidates any mapping established by the task's previous calls to `MapNuBus()`.

The C declaration for `MapNuBus()` is

```
char *MapNuBus ( ptr )
char *ptr;
```

The following example shows how to call `MapNuBus` in assembly language. Only the register supplied is modified. The address may be specified by an A register or a D register. The mapped address is returned in the register supplied.

```
MapNuBus A0
```

▲ **Caution** To move data across the NuBus, use `Netcopy()` (described next). Tasks that use the A/ROSE utility `MapNuBus()` must assume the responsibility for checking NuBus boundaries. Some hardware cards, including the MCP card, have a limited NuBus address space through which NuBus accesses are made. The hardware page latch that controls this NuBus address space needs to be changed whenever address boundaries are crossed. `Netcopy()` checks for and correctly handles these boundaries. ▲

The Macintosh IIci also has memory discontinuities that further complicate the use of MapNuBus.

NetCopy( )

NetCopy is a solution to many problems involving virtual memory. NetCopy exists in both A/ROSE running on a smart NuBus card and the A/ROSE Prep driver running on the Macintosh main board.

NetCopy takes two virtual addresses for its address parameters. NetCopy then examines internal A/ROSE structures to determine if it can convert the virtual addresses to NuBus addresses. These internal structures are initizlized when A/ROSE on the smart NuBus card and the A/ROSE Prep driver on the Macintosh main board initialize. The internal structures are then updated if LockRealArea or UnlockRealArea services are called on the Macintosh main logic board in a virtual memory environment.

If NetCopy cannot convert a ***virtual*** address to a NuBus address, NetCopy sends an internal A/ROSE message to a task located in the appropriate virtual address space that can perform the conversion. An internal A/ROSE message is returned to NetCopy when the conversion is completed.

In a virtual memory environment, LockRealArea increases the execution speed of NetCopy. When converting virtual addresses to NuBus addresses, NetCopy examines the internal structures updated by LockRealArea and UnlockRealArea . NetCopy does not have to send internal A/ROSE messages if the buffers were locked down using LockRealArea.

◆ *Note:* NetCopy will never be as fast as using MapNuBus and doing the copying directly. However, NetCopy is much safer than using MapNuBus.

`NetCopy()` will copy data from a source virtual address to a destination virtual address. `NetCopy()` has been designed to be safe and convenient, although you are advised to use only `NetCopy` for your data transfers. If the memory areas specified are locked and frozen in memory, then the copy process will go very fast.

▲ **Caution** NetCopy may send messages and issue blocking receive requests to wait for replies. Therefore, `NetCopy` must not be called at interrupt level by code that must be run in run-to-block mode, or called by code on the tick or idle chain. ▲

The C declaration for `NetCopy()` is the following:

```
short NetCopy(tid_type srcTID, void *srcAddress,

              tid_type dstTID, void *dstAddress,

              long bytecount);
```

The virtual address `srcAddress` is the virtual address space of the task whose Task ID is `srcTID`. The virtual address `dstAddress` is the virtual address space of the task whose Task ID is `dstTID`.

`NetCopy()` will safely, perhaps slowly, copy data from the source to the destination. Both the source and destination virtual addresses can be paged out to disk in a virtual memory environment. `NetCopy()` will cause these pages to be brought back into physical memory and perform the copy. The copying of data might be done by the processor on the Macintosh main logic board rather than by the NuBus card.

`NetCopy()` returns zero if the copy was successful. Otherwise, `NetCopy()` returns an error status.

Error Codes:

Non zero if there was an error in `NetCopy()` .

▲ **Warning** `srcAddress` and `dstAddress` must both be 32-bit clean virtual addresses. Memory manager flags must not be in the high byte of a Macintosh main logic board address.

Do not call `NetCopy` from interrupt routine because it does a blocking receive. Do not call `NetCopy` in idle chain because you cannot block idle chain. ▲

If you call `NetCopy` from task that runs under run to block mode, be aware that `NetCopy` may do a receive and give up the control of the CPU.

Register_Task( )

`Register_Task()` allows a task to register itself with the object and type names specified, using the Name Manager. The object and type names must not exceed 32 characters. If the task should be visible only to other tasks on the same card, `local_only` is set non-zero. If the task should be seen by other tasks on other cards, then `local_only` is set to 0. `Register_Task()` returns a non-zero value if the task was registered; otherwise, 0 is returned.

▲ **Warning** `Register_Task()` communicates with the Name Manager and issues a blocking `Receive`; therefore, the application gives up control of the CPU during this call. Do not use the register-task utility in code that can be called at interrupt level, such as a completion routine, or the tick or idle chain. ▲

The C declaration for `Register_Task()` is

```
typedef boolean short;
char     Register_Task ( object, type, local_only)
char     object [];
char     type [];
boolean  local_only;
```

The following code provides an example of how to register a task:

```
if (!Register_Task ("my_name", "my_type", 0))
   printf("Could not Register Task");
```

The following example shows how to call the `Register_Task` routine in assembly language:

```
MOVE.L      #LOCAL, -(A7)       ; value of local on stack
PEA         TYPE_NAME           ; address of type name
PEA         OBJECT_NAME         ; address of object name
JSR         Register_Task
ADDQ.W      #12,A7              ; pop the stack
TST.B       DO                  ; check if register ok
BNE.S       OK                  ; jump if OK
```

AROSESecs2Date( )

AROSESecs2Date() takes the number of seconds elapsed since 12:00 P.M. (Midnight), January 1, 1904, as specified by the seconds parameter, converts it to the corresponding date and time, and returns the corresponding date/time record in the date parameter.

The C declaration for AROSESecs2Date() is

```
pascal void AROSESecs2Date(secs, Date)
       long          secs;
       DateTimeRec   *Date;
       extern;
```

The following example shows how to call AROSESecs2Date from assembly language:

```
       Move.L        secs, -(A7)          ; number of seconds
       PEA           Date          ; Address for result -
                                   ; date/time record
       JSR           Secs2Date
```

Refer to the utility AROSEDate2Secs() earlier in this chapter for an example program that shows how to use each date/time request.

SwapTID( )

SwapTID() swaps the mFrom and mTo fields of a message buffer.

The C declaration of SwapTID() is

```
void SwapTID( mptr );
message    *mptr;      /* pointer to message buffer */
```

The form for the SwapTID macro is as follows, where P1 is the address of the message buffer:

```
[Label]        SwapTID        P1
```

P1 can be specified as a register (A0-A6, D0-D7), or can use any 68000 addressing mode valid in an LEA instruction to specify the location containing the desired address.

ToNuBus( )

`ToNuBus()` translates the pointer into a format suitable for passing to processes that may be on other cards. The pointer may contain a local address, which is translated to a NuBus address. `ToNuBus()` passes through 0 and NuBus addresses without modification.

◆ *Note:* Addresses on the MCP card are already in NuBus address form. This call is included to provide functionality for future releases.

The C declaration for `ToNuBus()` is

```
char *ToNuBus ( ptr )
char *ptr;
```

The following example shows how to call `ToNuBus` from assembly language. Only the specified register is modified **<<Reviewers, is this correct?>>**. The NuBus address may be specified by an A register or a D register, or through any other 68000-addressing mode (other than auto-increment or auto-decrement). The NuBus address is returned in the register or location supplied.

```
ToNuBus A0
```

TraceReg( )

`TraceReg()` is used to register the current task as the Trace Manager. For more information, refer to the section on the Trace Manager in Chapter 6.

The C declaration for `TraceReg()` is

```
void TraceReg ();
```

The following example shows how to call `TraceReg()` in assembly language:

```
JSR TraceReg
```

Chapter 6 **A/ROSE Managers**

THIS CHAPTER describes the operating system managers provided with A/ROSE. A manager is a task that provides a set of services to other tasks; each manager is specific to a certain function. The description of each manager includes the message codes used by that manager. ■

Table 6-1 lists the managers provided with the A/ROSE operating system and a brief description of each.

■ **Table 6-1** A/ROSE Managers

Name	Description
Echo Manager	Returns messages sent to it. Useful for diagnostic purposes, and as a mechanism to time messages between cards or between machines
InterCard Communications Manager	Responsible for intercard message delivery and transport (sending and receiving all messages between cards)
Name Manager	Provides naming services to tasks
Print Manager	Provides a means to print and to display information and debugging messages
Remote Systems Manager	Executes system calls on behalf of tasks on other cards
Timer library	Provides timing services to tasks
Trace Manager	Sends copies of all messages to a Trace Monitor (if available) for debugging purposes

Note: The Timer Manager is provided in this version of the A/ROSE software for compatibility with previous versions; it may not appear in the next version.

Echo Manager

The Echo Manager returns each message it receives to the sender. The Echo Manager is primarily used in the early stages of development:

- test messaging
- determining how long the IPC takes to send a message round-trip to a card or the Macintosh II

The Echo Manager operates with a single message loop. For each message it receives, it first checks if the received message is marked as undeliverable. If so, it is a message the Echo Manager already attempted to send the message and it is discarded, If not, the Echo Manager increments the message code, sets the message destination to the previous source of the message, sets the message source to the TID of the Echo Manager, and sends the message.

InterCard Communications Manager

The InterCard Communications Manager (ICCM) sends and receives all messages between cards and provides a mechanism tasks use to find out which other cards are configured on the NuBus.

◆ *Note:* Slot 0 has an implicit ICCM, since the ICCM is built into the A/ROSE Prep driver that is configured into the System File of the Macintosh II.

At initialization time, the ICCM on a smart card registers itself with the operating system; the task identifier of ICCM may be found by using `GetICCTID()`, described in Chapter 5.

ICCM then attempts to discover if any other smart card installed (including slot 0) has an ICCM running by searching the RAM of the card for the ICCM area. If it is found, the ICCM area writes the NuBus address of its own communication area to the corresponding ICCM. This action makes the receiving ICCM aware of the startup of a new ICCM on the other card that it missed at its own initialization time.

ICC_GETCARDS

`ICC_GETCARDS` is a message code to the ICCM that allows a task to find out which other cards are known by ICCM on the NuBus. Conditionally, `ICC_GETCARDS` also allows a task to find the TID of the Name Manager on each of the configured cards. The `ICC_GETCARDS` message is passed with a buffer of size (`struct ra_GetCards`). Each entry is filled in by ICCM, with the status of the card installed in the corresponding slot and, optionally, with the TID of the Name Manager on that card. The buffer contains one entry per slot number.

The message parameters for `ICC_GETCARDS` are as follows:

`mCode`	`ICC_GETCARDS`
`mDataPtr`	`Pointer to a data buffer`
`mDataSize`	`Length of data buffer`

Remember, the convention within A/ROSE is that an even `mCode` is a request and an odd `mCode` is a reply. For example, the ICCM request code `ICC_GETCARDS(150)` is even; the ICCM reply code `ICC_GETCARDS+1(151)` is odd. The Name Manager request code `NM_REG_TASK(100)` is even; the Name Manager reply code `NM_REG_TASK+1` is odd.

The data buffer format for ICC_GETCARDS is

```
#define IC_MaxCards 16;    /* Maximum NuBus Cards */
struct ra_GetCards
{
tid_type tid [IC_maxcards];
};
```

Each entry in the tid array corresponds to a NuBus slot number (tid[0] is slot 0, tid[1] is slot 1, and so on). ICCM fills in each entry with the information shown in *Table 6-2.*

■ **Table 6-2** Card status

Value of the entry	Card status
<0	Either does not exist or has no functioning ICCM
=0	Exists, and has an ICCM but no Name Manager
>0	Exists, and has an ICCM; this value is the Name Manager's TID

The returned TID may be used in the mTo field of a message to send a message to the Name Manager on the card corresponding to the entry.

Name Manager

The Name Manager performs functions similar to those of Name Binding Protocol (NBP) in AppleTalk. Tasks can register and unregister their names, look up the task identifiers of named tasks, and look up the name of a task corresponding to a given task identifier. The Name Manager allows tasks to become visible to other tasks on the same card and, optionally, to tasks on other cards.

The messages passed to the Name Manager are listed and described in *Table 6-3.*

■ **Table 6-3** Name Manager message codes

Name	Description
NM_LOOKUP_NAME	Looks up all object and type names for specified tasks
NM_LOOKUP_TID	Looks up the task identifiers for specified Type Names and Object Names
NM_N_SLOT_REQ	Provides notification of communications loss
NM_N_SLOT_CAN	Cancels the request for notification of communications loss
NM_N_TASK_REQ	Provides notification of task termination
NM_N_TASK_CAN	Cancels the request for notification of task termination
NM_REG_TASK	Registers the task name
NM_UNREG_TASK	Unregisters the task name

A task has two names: a type name and object name. Each name is a maximum of 32 characters long. (The MCP implementation of type name and object name is similar to Inside *AppleTalk*. For more information on type names and object names, refer to *Inside AppleTalk*.)

◆ *Note:* Any character may be used in a Type or Object name; however, the equal sign (=), a wildcard character, should be avoided since it is not possible to match it explicitly.

The parameters in the message buffer, that are sent to the Name Manager to look up names, look up task identifiers, and register tasks, are passed in a data buffer associated with the message buffer. The address of the buffer is placed in the message field `mDataPtr`, and the size of the buffer is placed in the message field `mDataSize`. The message to unregister a task contains in the `mFrom` field the task identifier of the task to unregister.

The following structures (defined in the file `managers.h`) are used when calling the Name Manager:

```
struct obj_rec                              /* object name record */
{

      utiny o_len;                          /* length of object name */
      char o_name [NM_Obj_Size_Max];        /* object name */
};

struct typ_rec                              /* type name record */
{
      utiny t_len;                          /* length of object name */
      char t_name [NM_Type_Size_Max];       /* type name */
};

struct pb_register_task                     /* register name param block */
{
      struct obj_rec rt_on;                 /* object name */
      struct typ_rec rt_tn;                 /* type name */
      char rt_local_vis;                    /* locally visible only flag */
};

struct ra_ltid                              /* return area for lookup tid */
{
      struct obj_rec ra_on;                 /* object name */
      struct typ_rec ra_tn;                 /* type name */
      tid_type ra_tid;                      /* task id */
};

struct pb_lookup_tid /* lookup task id parameter block */
{
      struct obj_rec ltid_on;               /* object name */
      struct typ_rec ltid_tn;               /* type name */
      unsigned short ltid_index;            /* index */
      unsigned short ltid_RAsize;           /* size of return area */
```

```
     struct ra_ltid ltid_ra [1];          /* return area (OUTPUT) */
};

struct ra_lnm                             /* return area for lookup name */
{
     struct obj_rec ra_on;                /* object name */
     struct typ_rec ra_tn;                /* type name */
};

struct pb_lookup_name
{
     tid_type lnm_tid;                    /* task id */
     unsigned short lnm_index;            /* index (INPUT/OUTPUT) */
     unsigned short lnm_RAsize;           /* size of return area */
     struct ra_lnm lnm_ra [1];            /* return area (OUTPUT) */
}
```

The Name Manager registers itself with Object Name `name manager` and Type Name `name manager`. The Name Manager is found by calling `GetNameTID()`, or by sending ICCM an `ICC_GETCARDS` message.

Looking up tasks

You can look up tasks by name or task identifier, by using one of the Name Manager messages:

- `NM_LOOKUP_NAME`
- `NM_LOOKUP_TID`

NM_LOOKUP_NAME

`NM_LOOKUP_NAME` returns all Object Names and Type Names for the specified task identifier. If no task identifier was found, then the size of Object Name will be set to zero. The index parameter (in the parameter block) on the initial call must be set to zero.

The parameter block for `NM_LOOKUP_NAME` is as follows:

```
struct pb_lookup_name
{
      tid_type         lnm_tid;          /* task id */
      unsigned short   lnm_index;        /* index (INPUT/OUTPUT) */
      unsigned short   lnm_RAsize;       /* size of return area */
      struct ra_lnm    lnm_ra [1];       /* return area (OUTPUT) */
};
```

The return area specified will be filled with zero, or with one or more entries of the following form:

```
struct ra_lnm                            /* return area for lookup name */
{
      struct obj_rec ra_on;              /* object name */
      struct typ_rec ra_tn;              /* type name */
};
```

The last entry plus one (entry+1) in the return area has the length of Object Name set to zero to indicate that there are no more entries to follow. If the return area is not large enough to hold all entries that could be returned, the index is set to a non-zero value. A subsequent NM_LOOKUP_NAME message must be sent to retrieve these entries, with the value of index set to the returned value of the previous NM_LOOKUP_NAME message.

The minimum size of the return area must be large enough to hold at least one entry plus the size of Object Name. To return more information, increase the size enough to hold the number of entries that the requesting task requests to process.

The parameter block for NM_LOOKUP_NAME is as follows:

```
struct  pb_lookup_name
{
        tid_type          lnm_tid;          /* task id */
        unsigned short    lnm_index;        /* index (INPUT/OUTPUT) */
        unsigned short    lnm_RAsize;       /* size of return area */
        struct ra_lnm     lnm_ra [1];       /* return area (OUTPUT) */
};
```

The message parameters for NM_LOOKUP_NAME are as follows:

```
        mCode         NM_LOOKUP_NAME
        mDataPtr      Address of the parameter block
        mDataSize     Size of the parameter block
```

NM_LOOKUP_TID

NM_LOOKUP_TID looks up the task identifiers of all tasks that match the Type Name and the Object Name specified. Use the equal sign (=), a wildcard character, to match all names. The index parameter on the initial call must be set to zero.

The parameter block for NM_LOOKUP_TID is as follows:

```
struct  pb_lookup_tid                       /* lookup task id parameter block */
{
        struct obj_rec    ltid_on;          /* object name */
        struct typ_rec    ltid_tn;          /* type name */
        unsigned short    ltid_index;       /* index */
        unsigned short    ltid_RAsize;      /* size of return area */
        struct ra_ltid    ltid_ra [1];      /* return area (OUTPUT)*/
};
```

The return area specified will be filled with zero or with one or more entries of the form:

```
struct  ra_ltid                             /* return area for lookup tid */
{
        struct obj_rec    ra_on;            /* object name */
        struct typ_rec    ra_tn;            /* type name */
        tid_type ra_tid;                    /* task id */
};
```

The last entry plus one (entry+1) in the return area has the length of Object Name set to zero to indicate that there are no more entries to follow. If the return area is not large enough to hold all entries that could be returned, A/ROSE sets the index to a non-zero value. You must make a subsequent `NM_LOOKUP_TID` message to retrieve these entries with the value of index set to the returned value of the previous `NM_LOOKUP_TID` message.

The return area must be large enough to hold at least one entry, plus the size of Object Name. For more information to be returned, the size should be increased to hold the number of entries that the requesting task attempts to process.

The message parameters for `NM_LOOKUP_TID` are as follows:

mCode	NM_LOOKUP_TID
mDataPtr	Address of the parameter block
mDataSize	Size of the parameter block

Notification of Communications Loss

A task can request that the Name Manager notify it when a card in a slot changes its communications status. The Name Manager immediately replies to the request, indicating whether the card in the slot is "up" or "down". The card is defined to be "up" if A/ROSE is running on that card. The Name Manager continues to notify the task whenever the status of the card in the slot changes until the task either

- stops running, or
- issues a request to the Name Manager to cancel notification of communications status for that card slot

NM_N_SLOT_REQ

The Notification of Communications Loss request must be sent to the Name Manager on the card where the requesting task is running. The message parameters for Notification of Communications Loss are as follows:

mCode	NM_N_SLOT_REQ
mOData[0]	Card slot number. Slots are numbered from 0x00 through 0x0f.

◆ *Note:* The Macintosh II currently supports slot 0, as well as slots 0x09 through 0x0e.

The reply parameters for Notification of Communications Loss are as follows:

mCode	NM_N_SLOT_REQ+1	
mStatus	NM_NO_ERRORS	If the card in the slot is up
	NM_SLOT_NOT_UP	If the card in the slot is down

NM_N_SLOT_CAN

The message parameters for Cancel Notification of Communications Loss are as follows:

mCode	NM_N_SLOT_CAN
mOData[0]	Card slot number. Slots are numbered from 0x09 through 0x0e.

◆ *Note:* The Macintosh II currently supports slot 0, as well as slots 0x09 through 0x0e; the value -1 specifies all slots.

The reply parameters for Cancel Notification of Communications Loss are as follows:

mCode	NM_N_SLOT_CAN+1	
mStatus	NM_NO_ERRORS	Request processed.
	NM_NO_ENTRY_FOUND	The task has no communications loss requests.

Notification of Task Termination

A local task can request that a remote Name Manager notify it when a task on the Name Manager's card is terminated. The Name Manager immediately replies to the request, indicating whether the remote task is currently running or not. The remote task is considered to have terminated if it stops or if it issues an NM_UNREG_TASK request.

◆ *Note:* The Name Manager must be running on the slots of both the remote task and the local task.

NM_N_TASK_REQ

The message parameters for Notification of Task Termination are as follows:

mFrom	TID	Task Identifier of the requesting or local task
mCode	NM_N_TASK_REQ	
mOData[0]	TID	Task Identifier of the remote task to monitor

The reply parameters for Notification of Task Termination are as follows:

mCode	NM_N_TASK_REQ+1	
mStatus	NM_NO_ERRORS	if the remote task is currently running
	NM_TASK_NOT_EXIST	if the remote task is not running
	NM_NAME_NOT_REG	if there is no Name Manager on the card where the local task is running

NM_N_TASK_CAN

The message parameters for Canceling Notification of Task Termination are as follows:

mFrom	TID	Task Identifier of the local task
mCode	NM_N_TASK_CAN	
mOData[0]	TID	Task Identifier of the remote task to monitor. The value of -1 specifies that all notification of task termination requests by this local task be cancelled.

The reply parameters for Canceling Notification of Task Termination are as follows:

mCode	NM_N_TASK_CAN+1	
mStatus	NM_NO_ERRORS	Request processed.
	NM_NO_ENTRY_FOUND	The local task had no outstanding request for notification of termination of the remote task.

Registering tasks

You can register and unregister tasks by using one of these Name Manager messages:

- NM_REG_TASK
- NM_UNREG_TASK
- NM_UNREG_NAME

NM_REG_TASK

NM_REG_TASK allows a task to become visible either to tasks on the local card only or to all tasks in the system. If rt_local_vis is non-zero, then this task is not visible to Lookup Task ID utility, NM_LOOKUP-NAME, or NM_LOOKUP_TID messages from other cards. Tasks may only register with the Name Manager on their own card. If the name is already taken, the error NM_DUPLICATE_NAME is returned in the message field mStatus.

The parameter block for NM_REG_TASK is

```
struct pb_register_task                          /* register name param block */
{
      struct obj_rec rt_on;              /* object name */
      struct typ_rec rt_tn;              /* type name */
      char       rt_local_vis;           /* locally visible only flag */
};
```

The message parameters for NM_REG_TASK are as follows:

mCode	NM_REG_TASK
mDataPtr	Address of the parameter block
mDataSize	Size of the parameter block

NM_UNREG_TASK

NM_UNREG_TASK removes all entries in the Name Table for the task issuing the call. When a task terminates, any names it had will be removed automatically.

The message parameters for NM_UNREG_TASK are as follows:

```
mCode         NM_UNREG_TASK
mDataPtr      0
mDataSize     0
```

NM_UNREG_NAME

Another Name Manager allows a task to unregister a single name. Any other registered names for this task are *not* affected. Unlike the Unregister Task request, Task Termination messages are *not* sent as a result of this request.

The following is example code using the new Name Manager request to unregister a single name. It is assumed that the name was previously registered by this task.

Following is the new structure defined for unregister name request:

```
struct pb_unregister_name
{
      struct obj_rec ut_on;
      struct typ_rec ut_tn;
};


#include    "os.h"
#include    "managers.h"

      struct  pb_unregister_name  *lunr_ptr,  punr_buffer;

      lunr_ptr  =  &punr_buffer;


      if  ((p  =  GetMsg  ())  ==  NULL)

            illegal ();


      p  ->  mCode  =  NM_UNREG_NAME;
```

```
p -> mDataPtr = (char *) lunr_ptr;

p -> mDataSize = sizeof (struct pb_unregister_name);

p -> mTo = GetNameTID();

strcpy(lunr_ptr -> ut_on.o_name, "Name1");


lunr_ptr -> ut_on.o_len = sizeof ("Name1") - 1;

        /* Do not include zero byte in length! */


strcpy(lunr_ptr -> ut_tn.t_name, "pmr");

lunr_ptr -> ut_tn.t_len = sizeof ("pmr") - 1;

        /* Do not include zero byte in length! */


Send(p);
```

When the reply is received, `mStatus` indicates the success or failure of the request. The following table lists each `mStatus` option with the corresponding meaning.

`mStatus`	Meaning
NM_NO_ERRORS	The request completed successfully.
NM_NAME_NOT_FOUND	The name was not registered.
NM_RT_PB_TOO_SMALL	The parameter block is too small
NM_RT_REMOTE_REG	The request was sent to a Name Manager for which the task is not local

Printing support

Printing is accomplished by using the library `printf` code and the Print Manager.

Each time `printf` is called and does not know the TID of the Print Manager, it searches for a Print Manager starting at slot 0, and continues searching the remaining slots until a Print Manager is found or all the slots have been searched. If printf knows the TID of the Print Manager and a Print Manager is found, the printf code sends the text to the Print Manager.

▲ **Caution** If the Print Manager is not found after thirty seconds, the text is discarded with no indication to the calling code. ▲

The printf code is linked into the user task; you install the Print Manager on a card or on the Macintosh II. (Refer to osmain for an example of using the print manager on a card; see pr_manager in the Apple IPC example folder for the Macintosh II).

After receiving a message from printf, the Print Manager code sends the contents of the message to the print device, and sends a reply to the requesting task's printf code when the information in the buffer has been printed. The Print Manager call includes the print buffer request, PRINT_ME, described next.

Print Manager operates with a single message loop. For each output request message it receives, Print Manager outputs as specified in the message and sends a reply when the message has been printed or discarded.

Table 6-4 lists the standard conversion characters that the printf code supports.
Table 6-5 lists the nonstandard conversion characters that printf also supports.

■ **Table 6-4** `Printf` standard conversion

Characterx	Standard conversion
`%d`	decimal conversion
`%u`	unsigned conversion
`%x`	hexadecimal conversion
`%X`	hexadecimal conversion with capital letters
`%o`	octal conversion
`%c`	character
`%s`	string
`%m.n`	field width, precision
`%-m.n`	left adjustment
`%0m.n`	zero padding
`%*.*`	width and precision taken from arguments

Note: `Printf` does not support `%f`, `%e`, or `%g`. It accepts, but ignores, a `'l'` as in `%ld`, `%lo`, `%lx`, and `%lu`.

■ **Table 6-5** `Printf` nonstandard conversion

Character	Nonstandard conversion
`%b`	binary conversion
`%r`	roman numeral conversion
`%R`	roman numeral conversion with capital letters

The Print Manager registers itself with Object Name `"print manager"` and Type Name `"print manager"`. The Print Manager slot is determined by the Start Parameters specified in `osmain`.

Print Buffer request

The Print Buffer request allows a task to specify a buffer that contains data to be printed. The message parameters for the Print Buffer request are as follows:

```
mCode          PRINT_ME
mDataPtr       Pointer to data buffer
mDataSize      Length of data (in bytes)
```

◆ *Note:* Applications do not normally need to directly use Print Manager. The `printf` code implements Print Manager interface on behalf of the application.

Remote System Manager

The Remote System Manager (RSM) on a remote card is responsible for executing system calls on behalf of local tasks. The local task sends a message to the Remote System Manager on a remote card specifying the desired request; the request is processed and the result is returned to the local task.

The Remote System Manager supports the following functions:

- `RSM_FreeMem`
- `RSM_GetMem`
- `RSM_StartTask`
- `RSM_StopTask`

The Remote System Manager registers itself with Object Name `"RSM"` and Type Name `"RSM"`. The Remote System Manager is found by using the `Lookup_Task` utility.

The end of this section mentions how to find out information regarding the Remote System Manager on another card and how to use the Remote System Manager on other card to control tasks on that remote card.

RSM_FreeMem

`RSM_FreeMem` returns the memory specified to the free pool. The memory must have been previously obtained on the destination card by using either the `GetMem()` system primitive or the `RSM_GetMem` message. The calling parameter `mDataPtr` contains the virtual (local) address of the memory to be released.

The calling parameters for `RSM_FreeMem` are as follows:

`mCode`	`RSM_FreeMem`
`mDataPtr`	`virtual (local) address of the memory to be released`

The reply parameters for `RSM_FreeMem` are as follows:

`mCode`	`RSM_FreeMem + 1`
`mDataPtr`	`Original pointer if mStatus != RSE_NO_ERRORS; otherwise, 0`
`mStatus`	`RSE_NO_ERRORS if memory buffer released`
`mStatus`	`RSE_NOT_MEM if not a memory buffer`

▲ Caution In most cases, A/ROSE on the remote card executes an illegal instruction if an attempt is made to free a memory buffer that has not been allocated by A/ROSE. ▲

When issuing the request to RSM_FreeMem, `mDataPtr` must contain the virtual address of the memory to be freed.

◆ *Note:* When A/ROSE is running on MCP cards or AST-ICP cards, the virtual address for memory on the card is the same as the NuBus address.

RSM_GetMem

RSM_GetMem obtains the memory specified from the free pool on the remote card. Two buffer addresses are returned to the caller if the buffer was allocated. The calling parameter mDataPtr contains the global (NuBus) address of the memory; the calling parameter mOData[0] contains the address of the memory on the remote card.

The calling parameters for RSM_GetMem are as follows:

mCode	RSM_GetMem
mOData[0]	Size in bytes (as in the GetMem primitive)

The reply parameters for RSM_GetMem are as follows:

mCode	RSM_GetMem + 1
MDataPtr	Address of buffer (as returned to RSM), or 0 if not allocated
mOData[0]	Global (NuBus) address of the buffer, or 0 if not allocated
mStatus	RSE_NO_ERRORS

When issuing the reply from RSM_GetMem, mOData[0] contains the NuBus address of the memory obtained and mDataPtr contains the virtual address of the memory obtained.

▲ **Warning** Be aware that in 1.1, the values returned in MOData[0] and MDataPtr are reversed from what happened in 1.0. The current values are stated earlier under reply parameters for RSM_GetMem. ▲

RSM_StartTask

RSM_StartTask creates a task and makes it eligible for execution on the remote card.

The calling parameters for RSM_StartTask are as follows:

mCode	RSM_StartTask
mDataPtr	struct *ST_PB; /* see StartTask primitive*/
mDataSize	sizeof (struct (ST_PB))

The reply parameters for RSM_StartTask are as follows:

mCode	RSM_StartTask + 1
mOData[0]	Task identifier of started Task or zero; if a Task identifier of zero was returned, an error may have occurred.

The parameter block for RSM_StartTask is the same as the operating system primitive StartTask().

◆ *Note:* The memory allocated for the code, data, and StartParameter segments must have been previously obtained on the remote card by a call to RSM_GetMem or GetMem().

RSM_StopTask

RSM_StopTask stops the task whose task identifier is specified, provided the task is running on the remote card.

The calling parameters for RSM_StopTask are as follows:

mCode	RSM_StopTask
mOData[0]	Task identifier of task to stop

The reply parameters for RSM_StopTask are as follows:

mCode	RSM_StopTask + 1
mStatus	RSE_NO_ERRORS

△ **Important** If one task stops another task, the one being stopped will not have the opportunity to release any message buffers that it is currently processing. . △

Finding the Remote System Manager

Tasks can determine the task identifier of a Remote System Manager on another card when you follow these steps:

1. Send an ICC_GETCARDS message to ICCM to obtain the task identifiers of the Name Managers on each of the known cards.
2. Use the Lookup Task utility to each found Name Manager, specifying the Object Name "RSM" and Type Name "RSM".

Loading remote tasks

Tasks may be loaded, started, and stopped on remote cards when you use the Remote System Manager on the remote card. To do so, refer to the file A/ROSE Prep:Examples:RSM_tester.c for annotated code.

◆ *Note:* If errors occur, then you must return any allocated memory to the card by sending a FreeMem message with the appropriate buffer to the Remote System Manager on the remote card.

The Remote System Manager processes the `RSM_StartTask` message, attempts to start the task, and returns either the task identifier of the started task or 0. If zero is returned or if errors are detected, then any allocated memory must be returned.

Timer library and Timer Manager

Both the timer library and the Timer Manager allow user programs to receive "wake-up" calls and to activate timing, cancel timing, set timing, and so forth. Timeouts are implemented as messages sent to the requesting tasks at specified times.

▲ **Caution** It is strongly recommended that you use the timer library rather than the Timer Manager, because the timer library provides greater performance and allows you to reliably cancel a timer when an event occurs. The Timer Manager is provided for compatibility with previous releases (primarily for using periodic timers without canceling timers), and will be removed in future versions of the software. ▲

Timer library

The timer library is available in the file `os.o` on the MCP distribution disk, and provides services similar to the Timer Manager.

The timer library handles timeouts for time-critical user code, and provides fast timer cancels and activations. You must use the include file `timerlibrary.h` in your code, which defines the interface to the calls listed in this section.

TLInitTimer( )

The `TLInitTimer()` call initializes the timer library, and must be the first call made to it. The parameter returned from `TLInitTimer` must be passed in all other timer library calls.

```
struct Tmem TOPB;
TOPB *TLInitTimer()
```

TLStartTimer( )

The `TLStartTimer()` call allows a task to request either a periodic or a one-shot timer message. The message is not available for use after the call.

◆ *Note:* Timer indication messages must be received through a `TLReceive()` call; they cannot be received by the primitive `Receive()` call.

```
char TLStartTimer (topb, m)
TOPB   *topb;
message       *m;
```

The message `m` must have been allocated and set up as a periodic or one-shot timer message as defined for the Timer Manager. TL`StartTimer` returns a non-zero value if the message was valid; otherwise `TLStartTimer` returns 0 and the message buffer may be reused or released by the calling task.

TLCancelTimer( )

The `TLCancelTimer` call allows the calling task to cancel a timer message. The timer message can be either a periodic or a one-shot timer message.

```
message *TLCancelTimer (topb, mID)
TOPB   *topb;
long   mID;
```

The canceled message matches the `mID` specified, unless the `mID` is zero. If the `mID` is zero, the first timer message to expire is canceled.

TLActiveTimer( )

The `TLActiveTimer()` call returns a count of the number of active timer messages.

```
long TLActiveTimer (topb, mID)
TOPB   *topb;
long   mID;
```

If `mID` is not zero, `TLActiveTimer()` returns 1; if the message corresponding to the `mID` is active, `TLActiveTimer()` returns 0; if the `mID` is zero, `TLActiveTimer()` returns the number of timer messages.

TLReceive( )

`TLReceive` is called to provide receive processing with timeout on behalf of the application.

```
message *TLReceive ( topb, mID, mFrom, mCode)
TOPB  *topb;
unsigned long       mID;
tid_type            mFrom;
unsigned short      mCode;
```

▲ **Caution** If you use the timer library, you must use the `TLReceive()` routine instead of the primitive `Receive()` request. ▲

`TLReceive` returns either the message that matches the `TLReceive` criteria or a timeout indication message (periodic reply or one-shot reply), whichever comes first.

Timer Manager

The Timer Manager provides timing services to tasks, and is useful when long timeouts are needed or where there is an infrequent need to start and cancel timers.

▲ **Caution** This section describing the Timer Manager is provided in this document for compatibility with previous releases. It is strongly recommended that you use the timer library. The Timer Manager may not be included in future releases. ▲

Table 6-6 lists the Timer Manager calls

■ **Table 6-6** Timer Manager calls

Calls	Description
Active Timer Query	Allows a task to determine if a particular timer is running or if any timers are running that are associated with the task
Cancel Timeout	Allows a task to cancel either an individual timer or all of the timers outstanding for the requesting task
Request One-Shot Timeout	Allows a task to receive a timeout reply *n* major ticks in the future
Requests Periodic Timeout	Allows a task to receive a periodic timeout reply starting *x* major ticks from when it is set, and then repeating every *y* major ticks

The user sends to the Timer Manager the desired timer message. The Timer Manager holds onto timeout request messages in its internal queue. A task may request either one-shot or periodic notification of timeout events.

- When a one-shot timeout occurs, the request is answered by returning to the user the original user message, with a message code of `TIMER_1_SHOT_REPLY`.
- When a periodic timeout occurs, the Timer Manager gets a message buffer from the operating system. This message buffer is returned to the user with a message code of `TIMER_PERIODIC_REPLY`. Any user data in the original message is copied into the message buffer that the Timer Manager uses for a reply.

Outstanding time events may be queried and, optionally, canceled. When the user requests that a timer be canceled, the original timer message is answered with a message status of timer canceled, followed by the response to the cancel-timer message.

◆ *Note:* Users should be careful in their use of message priority. A cancel message of a higher priority than the original periodic timeout request message could result in the cancel-timer reply arriving before the canceled timer message.

The number of ticks per second may be determined by calling the routine `GetTickPS()`.

The Timer Manager registers itself with Object Name `"timer manager"` and Type Name `"timer manager"`. You can find the task ID of the Timer Manager by calling `GetTimerTID()`, or by using the `Lookup_Task` utility.

Active Timer Query

Active Timer Query allows a task to determine if a particular timer is running or if any timers are running that are associated with the task.

The message code for the Active Timer Query is as follows:

```
TIMER_QUERY_REQUEST
```

The message parameters for the Active Timer Query are as follows:

`mOData[0]`	`Message ID`	If an individual timer is being queried
`mOData[0]`	`Zero`	If query is for any timer associated with the task

The reply-message code for the Active Timer Query is as follows:

```
TIMER_QUERY_REPLY
```

The reply parameters for the Active Timer Query are as follows:

`mOData[0]`	`Unchanged`
`mOData[1]`	`Number of timer messages found`

Cancel Timeout

Cancel Timeout allows a task to cancel either an individual timer or all of the timers outstanding for the requesting task. All outstanding timer messages are returned to the requesting task with a `TIMER_CANCELED` status.

The message code for Cancel Timeout is as follows:

```
TIMER_CANCEL_REQUEST
```

The message parameters for Cancel Timeout are as follows:

`mOData[0]`	`Message ID`	If an individual timer is to be canceled
`mOData[0]`	`Zero`	Cancel all timers associated with the task

The reply message code for Cancel Timeout is as follows:

```
TIMER_CANCEL_REPLY
```

The reply message parameters for Cancel Timeout are as follows:

`mOData[0]`	`Unchanged`
`mOData[1]`	`Number of timer messages canceled`

◆ *Note:* Users should be careful in their use of message priority. A cancel message of a higher priority than the original periodic timeout request message could result in the cancel-timer reply arriving before the canceled timer message.

Request One-Shot Timeout

Request One-Shot Timeout allows a task to receive a timeout reply a specified number of major ticks in the future.

The message code for Request One-Shot Timeout is as follows:

```
TIMER_1_SHOT_REQUEST
```

The message parameter for Request One-Shot Timeout is as follows:

`mOData[0]`	Time to wait in major ticks before replying

The reply message code for Request One-Shot Timeout is as follows:

```
TIMER_1_SHOT_REPLY
```

The reply message parameter for Request One-Shot Timeout is as follows:

`mOData[0]`	Unchanged

The possible error status for Request One-Shot Timeout is as follows:

```
TIMER_CANCELED
```

Request Periodic Timeout

Request Periodic Timeout allows a task to receive a periodic timeout reply starting a specified number of major ticks from when it is set, and then repeating at every specified interval thereafter.

The message code for Request Periodic Timeout is as follows:

```
TIMER_PERIODIC_REQUEST
```

The message parameters for Request Periodic Timeout are as follows:

`mOData[0]`	Time to wait in major ticks before first timeout reply
`mOData[1]`	Periodic interval in major ticks

The reply message code for Request Periodic Timeout is as follows:

```
TIMER_PERIODIC_REPLY
```

The reply message parameter for Request Periodic Timeout is as follows:

`mOData[0]`	Message ID of requesting user message

The possible error status for Request Periodic Timeout is as follows

```
TIMER_CANCELED
```

Trace Manager

The Trace Manager provides tracing services for messages sent between tasks, and includes calls to turn tracing on or off.

Upon startup, the Trace Manager waits to find a Trace Monitor registered with the Object Name `"Trace Monitor"` and Type Name `"Trace Monitor"`. No tracing is performed until a Trace Monitor is found that is so registered.

▲ **Caution** Once the Trace Manager registers, message throughput is dramatically reduced. When sending trace messages to the Trace Monitor, message throughput may be reduced by a factor of twenty or more, depending on the actions taken by the Trace Monitor. Even if tracing is turned off, the Trace Manager is still registered with the operating system and all messages must pass through it, reducing normal message throughput by more than half.

You cannot trace the Trace Manager. ▲

The Trace Monitor is an MPW tool that works with the Trace Manager to record all message traffic between tasks. The Trace Monitor relies on A/ROSE Prep to communicate with the Trace Managers on the cards; the Trace Monitor does little unless there are active Trace Managers present.

The format of the trace file is simply a sequence of messages. If a message has an associated data buffer (that is, `mDataSize` is non-zero), the message is immediately followed by the data buffer conents of size `mDataSize`. The syntax of the Trace Monitor command is

```
TraceMonitor [file]
```

where *`file`* is the name of the trace file in which to record message traffic. If *`file`* is not supplied, the default trace file name is `TraceFile`. The trace file is intended to be searched and interpreted by the MPW trace file dumping tool, DumpTrace, described later in this section.

Once a Trace Manager detects the presence of a Trace Monitor, the Trace Manager registers with A/ROSE using a `TraceReg` call and begins tracing. The A/ROSE `Send` primitive forwards all messages to the Trace Manager; the Trace Manager sends its own trace message to the Trace Monitor with the data pointer pointing to the traced message, and waits for an acknowledgement. The Trace Monitor records each traced message in a data file, along with any associated data, and acknowledges receipt of the message; the Trace Manager then forwards the original message to its intended destination. You can stop the Trace Monitor by pressing Command-period.

If the Trace Monitor fails to acknowledge in a reasonable time, the Trace Manager stops the process of sending trace messages to the Trace Monitor until it receives a message to turn tracing back on; this ensures that the message flow does not stop indefinitely. If necessary, the Trace Monitor can control tracing activity through the use of messages to the Trace Manager that direct it to turn tracing on or off.

Turn on tracing

The message code to turn on tracing is as follows:

```
TM_TRACE_ON
```

The Trace Manager assumes the request comes from the Trace Monitor, and uses the TID value of the message `mFrom` field as the TID of the Trace Monitor for subsequent tracing.

Turn off tracing

The message code to turn off tracing is as follows:

```
TM_TRACE_OFF
```

This stops the Trace Manager from sending trace messages to the Trace Monitor until tracing is turned back on.

Tracing messages

The trace message `TM_TRACE` describes the location of the traced message from the Trace Manager to the Trace Monitor. The message parameters for a trace message are as follows:

```
mCode        TM_TRACE
mDataPtr     Pointer to copy of traced message (and data)
mDataSize    Size of message plus size of data
```

The area pointed to by `mDataPtr` is a copy of the original message, immediately followed by the contents of the associated message data buffer (if any). The receiving message then has access to both the message and its data buffer.

The message code for acknowledging the receipt of a trace message to Trace Manager is as follows:

```
TM_TRACE+1
```

DumpTrace

DumpTrace is an MPW tool that searches and interprets message trace files created by the TraceMonitor tool. DumpTrace dumps the messages from each trace file specified. If you do not specify a file name, DumpTrace dumps the file `TraceFile`. The messages are dumped to standard output.

The syntax of DumpTrace is

```
DumpTrace  [-an][-cn][-dn][-fn][-in][-ln][-pn][-sn][-tn][file  ...]
```

where the following values are specified as hexadecimal numbers:

`-a` *n*	dump messages having `To` or `From` values of *n*
`-c` *n*	dump messages having `Code` value of *n*
`-d` *n*	dump messages having `DataPtr` value of *n*
`-f` *n*	dump messages having `From` value of *n*
`-i` *n*	dump messages having `ID` value of *n*
`-l` *n*	dump messages having `DataSize` value of *n*
`-p` *n*	dump messages having `Priority` value of *n*
`-s` *n*	dump messages having `Status` value of *n*
`-t` *n*	dump messages having `To` value of *n*
`file`	the name of the trace file in which to record message traffic

Messages are dumped selectively based on values specified by the options just listed. If options are specified, a message is dumped only if its fields match one of the values specified by each of the options. If no options are specified, all messages are dumped. Each option can be repeated with different values, as shown in the following examples.

Here is the first example:

```
DumpTrace -f0d000001 -f0d000002 -c64 FileName1 FileName2
```

In this example DumpTrace dumps from `FileName1` and `FileName2` those messages that have `Code` values of 100 (64 hex) and that are from task 0d000001 (slot d, task 1) or 0d000002 (slot d, task 2).

Here is another example:

```
DumpTrace -a0d000003
```

In this example, DumpTrace dumps from TraceFile those messages that are either to or from slot d, task 3.

The following example of DumpTrace shows output for a message with an associated data buffer:

```
To:       0d000001      Code:         0097          ID:           fd0009a1
From:     0d000005      Status:       0000          DataPtr:      0000090c
                        Priority:     0000          DataSize:     00000040
SData:  00  00  00  00  00  00  00  00  00  00  00  00 ............
OData:  00  00  00  00  fd  00  00  08  00  00  02  6c ...........1

0000090c  0000  0000  ffff  ffff  ffff  ffff  ffff  ffff  .............
0000091c  ffff  ffff  ffff  ffff  ffff  ffff  ffff  ffff  .............
0000092c  ffff  ffff  ffff  ffff  ffff  ffff  ffff  ffff  .............
0000093c  0c00  0001  0d00  0001  ffff  ffff  ffff  ffff  .............
```

Chapter 7 **Programming Notes for A/ROSE**

THIS CHAPTER describes methods to handle peculiarities of A/ROSE, and includes some guidelines and brief code examples for the following:

- accessing memory for intercard communications (including address mapping, intercard buffer copying, and intercard message passing)
- calling primitives from interrupt routines
- executing small routines at every major tick (using the Tick Chain)
- using the Idle Chain
- writing your own download program
- loading remote tasks ■

Intercard communications

Accessing memory that may be off-card introduces special coding considerations on cards using processors that do not directly support 32-bit addressing (such as the Motorola 68000). The MCP provides special hardware (page latch) to map off-card memory into the processor's address space.

Address mapping

You can use the `MapNuBus` function to set the hardware page latch and to return the appropriate local address. The operating system saves and restores the state of the hardware page latch (the address mapping) when task switching occurs. Interrupt routines that need to gain access to off-card buffers must also save and restore the state of the hardware page latch (the address mapping).

The following is an example that demonstrates a simple case of using `MapNuBus`.

```
message *mptr;
mptr = Receive(OS_MATCH_ALL, OS_MATCH_ALL, OS_MATCH_ALL, 0);
switch (mptr->mCode)
{
        case myCode:
                /* Process myCode */
                process_myCode(MapNuBus(mptr->mDataPtr));
                break;
        <<Other code >>
}
```

The function `process_myCode` processes the buffer associated with the message. Because `MapNuBus` was already called, it can simply treat the pointer it receives as an ordinary pointer, as long as the routine does not access any other off-card pointer or call `MapNuBus`.

▲ Caution To move data across the NuBus, use `NetCopy()` (described earlier in chapter 5) over `MapNuBus()`. The MCP card has a NuBus address space through which access to the NuBus is made. The hardware page latch that controls this NuBus address space needs to be changed whenever address boundaries are crossed; tasks which use `MapNuBus` may not check for these boundaries. However, `Netcopy()` checks for and correctly handles the boundaries. ▲

Intercard buffer copy

Any piece of code that manipulates more than one potentially off-card buffer at a time can be complex. For example, if you copy data between two such buffers the operating system will continually call `MapNuBus` to adjust the mapping hardware. This operation may actually be more efficient if the data is copied through an intermediate local buffer.

Intercard message passing

Normally, there is no need to be concerned about how messages are moved from one card to another, since A/ROSE handles these transparently to the use through the use of TIDs and ICCMs. However, this section is included to provide more detailed information about this function.

Communication between peer ICCMs is done by using the communication areas. The `Send()` primitive checks the `mTo` field of each message. If the `mTo` field specifies a destination that is not on the sender's card, the `Send` primitive passes the message unaltered to ICCM. ICCM then examines the `mTo` field to discover the destination of the message.

ICCM on the sending card first checks that any previous message in the communication area of the destination card has been processed. ICCM then checks that a new buffer is available to receive the message; if not, a new buffer is requested. When a buffer becomes available, ICCM writes into the communication area the message to be sent to the destination card.

The receiving ICCM polls the communication area for new messages. When a new message arrives, it is forwarded to the receiving task. Once the message has been forwarded, the receiving ICCM clears the sender's communication area on the receiver's card and supplies a new Receive buffer. The new buffer allows the sending ICCM to again send a message to the receiver's card.

If the destination does not exist, the message is returned to the sender as undeliverable. If the destination does exist, it is passed to a peer ICCM on the destination card. The ICCM on the destination card attempts to forward the message to the task specified. If the task does not exist, the message is returned to the sender as undeliverable.

VoidDCache()

As part of MCP 1.1, `VoidDCache()` has been added to the A/ROSE Prep driver to clear the data cache on a machine running with a Motorola 68030 processor. This routine is only effective if the processor is a 68030. This routine clears the 68030 data cache to prevent the 68030 processor from seeing stale data that was changed by a NuBus smart card or other DMA device.

The calling sequence in C is

```
void  VoidDCache();
```

The assembler language call is

```
        Import      VoidDCache
```

```
    JSR         VoidDCache
```

No result is returned.

◆ *Note:* The A/ROSE Prep driver voids the data cache automatically each time a message is received from a NuBus card and during vertical blank interrupts. (Applications do not have to call `VoidDCache` after receipt of a message from the card under one condition. The condition is if the applications examine buffers on the Macintosh main logic board that are shared with NuBus cards. The buffers should be examined only after receiving a message from a task on the card.)

Caching is totally disabled on the Macintosh IIci.

Interrupt handlers

This section describes some guidelines for calling primitives from interrupt routines.

When using interrupt routines, do not call the following primitives since results are unpredictable:

- `Receive()`
- `Reschedule()`

- `StartTask()`
- `StopTask()`

All other operating system primitives may be called from interrupt routines. However, be careful when using the primitives `GetMem()`, `FreeMem()`, and `Send()` because these primitives execute at the same interrupt level as the caller. This ensures that device-interrupt interlocks are maintained. `Send()` can be used to notify the appropriate task that a message has arrived; however, system performance may be impacted

Use of A/ROSE primitives at interrupt level should be minimized, because they may interfere with high-performance communication devices. User tasks should pre-allocate buffers for their interrupt routines, and should also release those buffers when the interrupt routine has finished with the buffer.

◆ *Note:* When using `GetMsg`, A/ROSE always fills in the `mFrom` field with the TID of the current user task. Your interrupt routine must overwrite the `mFrom` field with the task ID that will process any reply.

You can see an example of a task that uses interrupt routines to control hardware in the files `:A/ROSE :Examples:pr_manager.c` and `:A/ROSE :Examples:ossccint.a`. These files show how to control SCCs and use them in asynchronous mode.

The following is an example of how to install an interrupt routine, along with an example of an interrupt routine within the code:

```
IInstall     Proc   Export
      Import  PostRTE
      LEA          MyA5, A0                 ; Get address of location to hold A5
      MOVE.L       A5, (A0)                 ; Put A5 there for interrupt routine
      LEA          Lvl5, A0                 ; Get address of interrupt routine
      MOVE.L       A0, $74                  ; Put address of routine into vector
      RTS
MyA5  DC.L         0                        ; Holds A5 for interrupt routine

* Actual interrupt routine follows.
Lvl5  MOVEM.L      A0-A1/A5/D0-D2, -(A7)       ; Be sure to save
                                            ; registers

* If the routine is going to access the processes global data,
* A5 will have to be set to provide access.
      MOVEA.L      MyA5, A5                 ; Set A5 to this process' A5 value

<Do whatever you want here>

* If access to a possibly off-card buffer is needed,
* do something like this:
      MOVE.W       gCommon.gPageLatch, -(A7) ; Save page latch
      MapNuBus     A0                       ; Map address to access
```

```
<Access the buffer>
      MOVE.W       (A7)+, gCommon.gPageLatch  ; Restore page
      ResetLatch                        ; Reset mapping hardware to match

*Now get ready to leave the interrupt routine.
      MOVEM.L      (A7)+, A0-A1/A5/D0-D2   ; Restore registers
                                        ; saved on entry
      JMP          PostRTE              ; Return from exception
```

where:

- `gCommon.gPageLatch` contains the pagelatch value associated with the currently-executing task
- `ResetLatch` resets the hardware page latch based upon the value contained in `gCommon.gPageLatch`
- `PostRTE` provides a common exit routine from interrupt handlers

Tick Chain

The Tick Chain allows you to incorporate very small routines in the code that are executed at every major tick. For example, a Tick Chain routine might be the operating system allowing the ICCM to go out and look in buffers. Take care to ensure that shared data buffers are not touched by code placed in the Tick Chain; Tick Chain code is scheduled independently of A/ROSE tasks, including those in run-to-block mode.

The start of the Tick Chain is a location in low memory (`gTickChain`), which is a pointer to a subroutine that the timer interrupt code calls every major tick. The pointer allows the timer interrupt routine to call user-installed time-critical code routines. The number of ticks per second may be determined by calling the library routine `GetTickPS()`.

Register A5 is set up to allow access to A/ROSE global variables.

◆ *Note:* Any routine not loaded with the A/ROSE operating system that is placed in the Tick Chain/Idle Chain must use its own A5 value.

The routine in the Tick Chain/Idle Chain must preserve the value of A5 across the call and ensure that their routine is using the correct value of A5 during its processing. To do so, follow the steps listed below for the appropriate code:

In the code that inserts a routine into the Tick Chain/Idle Chain:

1. Save the value of A5 in the code segment for the routine in the Tick Chain/Idle Chain.
2. Save the address of the routine that is currently in the Tick Chain/Idle Chain.

3. Insert the address into the Tick Chain/Idle Chain.

In the routine in the Tick Chain/Idle Chain:

1. Save the value of A5.
2. Load the A5 value saved by your code segment that inserted this routine into the Tick Chain/Idle Chain.
3. Perform the desired operations.
4. Restore A5 to its previous value.
5. Call the routine that was saved in Step 2 of the first set of instructions (for the code that inserts the routine).

The following code segment shows how to install and use the Tick Chain mechanism:

◆ *Note:* Use this mechanism with caution, because it may degrade system performance unless you install extremely short time-duration code segments. To ensure that the operating system will reliably execute tasks and not hang the card, the total time of the routines installed should not exceed the duration set for the major tick.

```
void (*ticknextcall) ();
void tickinstall ()
{
       void          myRoutine ();
       extern        struct gCommon *GetgCommon ();
       short         s;
       struct        gCommon       *p;
       /* Fetch local of gCommon area    */
       p = GetgCommon ();
       /* disable interrupts      */
       s = Spl (7);
       /* Fetch next routine      */
       /* install myRoutine       */
       ticknextcall =  p -> gTickChain;
       p -> gTickChain = myRoutine;
       /* restore interrupts      */
       (void) Spl (s);
}

void myRoutine ();
{
/* please do something useful */
       ticknextcall ();
}
```

Idle Chain

The Idle task performs the following functions:

- increments a counter
- calls the Idle Chain
- issues the `Reschedule` primitive to allow other tasks to run

The Idle task runs in block mode, and is given the lowest priority (priority 0). When no other task is eligible for execution on the processor, A/ROSE schedules the Idle task.

The start of the Idle Chain is a location in low memory (`gIdleChain`), which is a pointer to a subroutine that the Idle task calls every time the Idle task is scheduled (`gIdleLoop` in `gCommonArea`). The pointer allows the Idle task to call user-installed, noncritical time-code routines. On entry, Register A5 is set to allow access to globals. Register A5 must be preserved across this call.

The following code segment shows how to install and use the Idle Chain mechanism.

◆ *Note:* Since the Idle task runs in block mode, use this mechanism with caution. The Idle Chain does not release control until the task is completed, and therefore can impact performance. You should install only extremely short time-duration code segments.

```
void (*idlenextcall) ();
void idleinstall ()
{
      void   myRoutine ();
      extern       struct gCommon *GetgCommon ();
      short  s;
      struct       gCommon    *p;
      * Fetch local of gCommon area */
      p = GetgCommon ();
      /* disable interrupts */
      s = Spl (7);
      /* Fetch next routine */
      /* install myRoutine */
      idlenextcall = p -> gIdleChain;
      p -> gIdleChain = myRoutine;
      /* restore interrupts */
      (void) Spl (s);
}

void myRoutine ();
{

/* very short time duration π shop rental calculator */
      idlenextcall ();
}
```

Writing your own download program <<Reviewers, does this section get deleted?>>

Two methods are available for downloading code onto NuBus cards. The first method is used during Macintosh startup when A/ROSE is not yet running on the card. In this case user code is downloaded onto the card with A/ROSE operating system code. To download A/ROSE and user code at the same time, the user must halt the card, download the code and the operating system, and then start the card. The second method is used to download only user tasks when the card is already running the A/ROSE operating system. In this case, the card must not be halted or started.

To implement one of these methods for downloading code onto NuBus cards, use one of the following download subroutines: `NewDownload` or `DynamicDownload`. NewDownload, described in the next heading, is a general download subroutine that can be used to download A/ROSE and user programs onto a card at the same time. `DynamicDownload` is a special subroutine that can be used to download user tasks onto cards running A/ROSE. These routines are supplied in an object library module named "download-lib.o" which also includes other subroutines such as `TestSlot`, `NewFindcard`, `StartCard`, and `HaltCard` (these routines are described later in this section.)

The new download subroutines read card-specific information from the A/ROSE Prep file, which must be in the System Folder of the startup volume. To control downloading, parameters can then be passed as arguments to either the `NewDownload` or `DynamicDownload` subroutines .

◆ *Note:* `NewDownload` neither stops nor starts the card. User programs must specifically issue any calls when using `NewDownload`. This gives users the option of downloading code and setting up user-specific data before starting the card.

The `NewDownload` or `DynamicDownload` subroutines provide special facilities to specify the following:

- the load address of the downloaded program
- the address of the gCommon area
- the type of the resource where the code can be found (default is 'CODE')
- the address offset to be used when accessing the card memory
- the start parameter segment address
- the length of the start parameter segment

The parameters you select depend on the subroutine you are calling. Operations such as initializing the `gCommon` area may be specified using the options parameter. The code may be downloaded when the card is already running A/ROSE by setting a bit in the options parameter. The start parameter segment may be specified when using `DynamicDownload`.

although `NewDownload` can be used to down load tasks dynamically, it is provided specifically for the initial download of A/ROSE system and manager tasks. You must specifically stop the card before downloading and start the card after downloading. Do not call `StartCard` or `HaltCard` when using `DynamicDownload`.

A Macintosh application is also included that can be used to download applications onto NuBus cards. The source for this application, (`ndld.c`,) is provided in the Downloader folder. `ndld.c` can be used as an example of how to call the `NewDownload` subroutine.

The Download subroutine and MPW tool download released with an earlier version of A/ROSE will not work with A/ROSE 1.1.

◆ *Note:* The resource file that contains the code to be downloaded must be the current resource file when `NewDownload()` or `DynamicDownload()` is called.

To use the download subroutines the user code must be linked with `:Downloader:download-lib.o`. The user programs can then call the download subroutines to download code onto NuBus cards in the Macintosh computer. The following sections describe the calling mechanisms for the download subroutines and the `ndld` application.

NewDownload

To use the `NewDownload` subroutine you must have the A/ROSE Prep file in the System Folder of the startup volume. The format of the A/ROSE Prep file will be described in a later tech note.

```
typedef pascal void (*PascalPtrLong)(long segSize);



pascal short NewDownload(slotNUM, addrOffset, loadaddr,

              gCommonAddr, options, restype, registers, ProgProc)



short  slotNUM;

/* Slot number to use when loading code. This parameter is NOT a bit
mask. Specify a value between 0x9 and 0xE inclusive. */



long   addrOffset;

/* Offset from address of card to use as start. The default  address is
defined by the symbol DEF_ADDROFFSET in file Download.h in the A/ROSE
includes folder */



long   loadaddr;
```

```
/* Address RELATIVE to addrOffset on the card to load data and code.
The default initial load address of A/ROSE is defined by the symbol
DEF_LOADADDR in file download.h in the A/ROSE includes folder.   */



struct gCommon *gCommonAddr;

/* Address RELATIVE  to addrOffset on the card to load locate the
gCommon area.  The  default initial load address of A/ROSE gCommon is
defined by the symbol DEF_GCOMMON in file Download.h in the A/ROSE
includes folder.   */



short  options;

/* Set to (DL_INITLOAD + DL_CLEARMEM) if an initial download and low
memory is to be cleared. Set to DL_INITLOAD if an initial download and
low memory is not to be cleared.  Set to 0 if a dynamic download. (Low
memory will not be cleared.) If an initial download is being done then
low memory, gCommon, and the jump tables will be set up.  If a dynamic
download is being done then low memory will NOT be touched.   */



ResType   restype;

/* The resource type (eg. 'CODE') of the resource to load into the
card.   */



struct ST_Registers  *registers;

/* Pointer to a register area (defined in os.h in the
":A/ROSE:includes:" folder) where the correct registers will be
returned for use in an RSM_StartTask request to start a task loaded
dynamically.  Users need to specify only an address of an area of
structure ST_Registers */



PascalPtrLong ProgProc;

/*  Progress report procedure.  Called with length of code segment
being downloaded.  This call is done before the segment is download.
This is a user written procedure that displays the download progress.
Specify 0 if you do not have a routine.   */
```

ProgProc is provided to facilitate continuous monitoring of the download process if you are downloading a large program. ProgProc is declared as a Pascal procedure that takes one long word as a parameter. Whenever the downloader is about to download a segment, the downloader calls the progress procedure with the size of the segment being downloaded.

Dynamically downloaded tasks that are downloaded by calling NewDownload are allocated a stack space of 4096 bytes and started with a priority of 10. The heap allocated is 0.

◆ *Note:* addrOffset must be 0 and gCommonAddr must be DEF_GCOMMON.

loadaddr must be greater or equal to DEF_LOADADDR.

Return status

The NewDownload routine can return any of the following error constants. The state of any NuBus cards to be downloaded is undefined if an error is returned. DLE_NOERR is a normal, successful return. The following is taken from the include file Download.h located in the folder :A/ROSE:includes:

```
/* Error Constants */

#define DLE_NOERR           0       /* No error                      */
#define DLE_NOJT            1       /* No jump table found           */
#define DLE_DATAINIT        2       /* Bad Data Init segment         */
#define DLE_GLOBALF         3       /* Global data format error      */
#define DLE_CODES           4       /* Code segment error            */
#define DLE_MAC2            5       /* Can only run on Mac II family */
#define DLE_EMPTY           6       /*  All slots are empty          */
#define DLE_NOCARD          7       /*  No card in specified slot    */
#define DLE_STARTERR        10      /*  Starting error               */
#define DLE_NOMEM           11      /*  No memory                    */
#define DLE_RSMERR          12      /*  RSM error                    */
#define DLE_NORSM           13      /*  No RSM                       */
#define DLE_NOAROSE         14      /*  No A/ROSE running on card    */
#define DLE_NORSRC          15      /*  No 'CNFG' resource           */
```

```
#define DLE_NOPREP          16      /*  No A/ROSE prep file          */
#define DLE_ABORT           17      /*  Aborting download            */


/* Useful constants */


#define Max_Slots           16      /* Max number of card slots */
```

▲ **Warning** Make sure the resource file is the top most resource and is open before making the call to NewDownload. The resource type of the resource to be downloaded is specified in restype. The caller is responsible to call HaltCard() and StartCard() to halt and start the hardware if you are downloading the subroutine for the first time. The support routines TestSlot(), NewFindCard(), StartCard, and HaltCard are provided to assist the user.. ▲

DynamicDownload

To use DynamicDownload you must have the A/ROSE Prep file in the System Folder of the startup volume; A/ROSE must also be active and running on the NuBus card you installed .

```
pascal tid_type DynamicDownload(slotNUM, restype,
       st_parmblock, startParmSegment, lenParmSegment)



short  slotNUM;

/* slot number to use when loading code. This is NOT a bit mask.
Specify a value between 0x9 and 0xE inclusive.  */



ResType   restype;

/* The resource type (eg. 'CODE') of the resource which has the code to
load into the card.  */



struct ST_PB *st_parmblock;

/* Pointer to an RSM_StartTask parameter area (defined in
:A/ROSE:includes:os.h). It is the caller's responsibility to initialize
the following fields:
```

```
st_parmblock->stack

st_parmblock->heap

st_parmblock->priority

st_parmblock->ParentTID


The updated parameter block along with the reply from Remote System
Manager is returned.  */



char *startParmSegment;

/* Starting parameters for the downloaded task.  */



long lenParmSegment;

/* Length of the startParmSegment  */
```

DynamicDownload returns the Task ID of the downloaded task. If the task could not be downloaded for any reason, a value of 0 is returned.

Supporting routines

The following routines (`TestSlot`, `NewFindCard`, `StartCard`, and `HaltCard`) are provided to help users administer basic functions to their NuBus cards.

TestSlot -

Use `TestSlot` to confirm that a NuBus card is capable of running A/ROSE. `TestSlot` checks a slot for a card with valid configuration information and returns the characteristics of the card.

```
short  TestSlot(slotNUM, boardIDPtr, lenMemPtr, startMemPtr,

                         CPURsrcPtr, networkRsrcPtr)



short  slotNUM;       /*  0x9 through 0xE.  Not a bit mask  */
short *boardIDPtr;  /* place where the board ID is returned */


long *lenMemPtr;     /* TestSlot returns the max. length of RAM in
location pointed by lenMemPtr  */


long *startMemPtr; /* TestSlot returns the starting address of the RAM
area in location pointed by startMemPtr  */


short CPURsrcPtr[4];       /*  CPURsrcPtr is a double long word area
where the CPU resource info is returned */


short networkRsrcPtr[4]; /*  networkRscrPtr is a double long word where
the network resource info is returned  */
```

The `BoardID` is a unique number assigned to each type of installed NuBus card. The starting address of RAM and the maximum length of RAM are returned in `startMemPtr` and `lenMemPtr`, respectively. The starting address of the RAM area is returned as a NuBus address and the length of the RAM is the maximum amount of RAM that the architecture of the board allows. The download subroutine finds the actual size of the RAM physically present in the card at the time of download. `CPURsrcPtr` and `networkRsrcPtr` are pointers to two arrays of four short words. `CPURsrc` and `networkRsrc` are returned to help the caller identify the type of NuBus card found.

`TestSlot` returns 0 if the slot does not contain a card capable of running A/ROSE. `TestSlot` returns -1 if the card is capable of running A/ROSE.

The collowing table lists Board IDs of some existing NuBus cards:

Board type	Board IDs (Decimal)
MCP board	10, 11, 13, and 25
ASIC MCP board	24,28, and 105
AST ICP board	261
Green Spring board	441

NewFindcard -

Use NewFindcard to find all smart cards that match the specified board ID installed in the system.

```
pascal  short  NewFindcard(slot,  boardID)

short  *slot;          /*  Where bit mask is returned  */

short  boardID;        /*  Specifies the type of the board  */
```

*slot is a bit mask indicating which slots are available for loading. Bit 0 is the least significant bit. Bit9 (bit mask 0x200) corresponds to slot 9. Bit 14 (bit mask 0x4000) corresponds to slot E.

Findcard will return DLE_NOERR if at least one card of the correct type is found. Findcard will return DLE_EMPTY if no card of the correct type is found.

If the specified boardID is zero, then NewFindcard will return the existence of all cards capable of running A/ROSE.

If the specified boardID is nonzero, then NewFindcard will find all the cards that match the boardID. The BoardID information is found in the configuration ROM of the card.

StartCard

Use StartCard to start the NuBus card

The StartCard subroutine routine resets the CPU on the NuBus card and starts the execution of the downloaded program. The download subroutine operates as if the card processor is of type Motorola 680X0; the subroutine loads the program counter into location 4 on the card and the stack pointer into location 0 on the card.

▲ **Warning** A halted or unloaded NuBus card should not be started without first downloading code into it. <<WHY NOT?>> ▲

```
short  StartCard  (SlotNum)

short  SlotNum;             /*  0x9 through 0xE  */
```

SlotNum contains the slot number of the card to start.

`Return status` represents the return values for `StartCard` routine are same as the `NewDownload` subroutine.

HaltCard -

Use `HaltCard` to To halt a specified NuBus Card.

The `HaltCard` subroutine halts the execution of programs on the NuBus card by activating the reset line. The card remains in the halted state until it is started again.

```
short HaltCard (SlotNum)

short SlotNum;              /*  0x9 through 0xE  */
```

`SlotNum` contains the slot number of the card to start or halt.

`Return status` represents The return values for `HaltCard` routines that are same as the `NewDownload` subroutine.

The following example (Compiled with MPW C 3.0) shows the two ways to download code onto NuBus cards.

To initially load and start a card in Slot 0x0D the `NewDownload` subroutine would be called in this way:

```
short slotNUM;

long addrOffset, loadaddr;

struct gCommon *gCommonAddr;

short options;

ResType restype;

struct ST_Registers registers;

PascalPtrLong progProc;

short refNum;

.

.
```

```
slotNUM= 0x0d;

addrOffset = DEF_ADDROFFSET;

loadaddr = DEF_LOADADDR;  /* 0x800 */

gCommonAddr = DEF_GCOMMON;  /* 0x400 */

options= DL_INITLOAD | DL_CLEARMEM;

progProc = NIL;

restype= 'CODE';


if ((refNum=OpenResFile("\pSampleCode")) != -1)

{

       if (options && DL_INITLOLAD)

       {

              if (HaltCard (slotNUM) != DLE_NOERR)

              {

                     /*  Cannot halt card  */


                     printf("\nError halt card in slot %x", slotNUM);

              }

       }

       else  /*  Not initial load  */

       {

              status = NewDownload (slotNUM, addrOffset, loadaddr,

                                    gCommonAddr, options, Resource,

                                     &registers, progProc);
```

```
        if (status != DLE_NOERR)
        {
            printf("\nDownload error in slot %x", slotNUM);
        }
        else  /* No error in downloading  */
        {
            if (options && DL_INITLOLAD)
            {
                if (StartCard (slotNUM) != DL_NOERR)
                {
                    /*  Cannot start card  */

                    printf("\nError start card in slot
%x",
    slotNUM);
                }
            }
        }
    }

    closeResFile (refNum);
}
```

To dynamically download a user task to the card in Slot 0x0D the `DynamicDownload` subroutine would be called in this way:

```
pascal tid_type DynamicDownload(slotNUM, restype,
       st_parmblock, startParmSegment, lenParmSegment)



short slotNUM;

struct ST_PB st_parmblock;

ResType restype;

char startParmSegment[] = {"-c 'ASDF' -t"};

                            /* start parameters for the user task */

long lenParmSegment;

short refNum;

tid_type taskID;

.

.

.



slotNUM= 0x0d;

restype= 'CODE';

lenParmSegment= sizeof startParmSegment;

st_parmblock.stack = 0x1000;

st_parmblock.heap = 0;

st_parmblock.priority = 10;

st_parmblock.ParentTID = 0;


if ((refNum=OpenResFile("\pSampleCode")) != -1)

{
       taskID = DynamicDownload (slotNUM, restype,
       &st_parmblock, startParmSegment,
       lenParmSegment);

       if (taskID != 0)

       {

              /*  You can send and receive messages to this task */



       .

       .

       .

       }

       .

       .

       .

       closeResFile (refNum);

}
```

Chapter 8 **Developing Smart Card Applications**

THIS CHAPTER describes how to develop software applications for the MCP smart card, and includes information on

- how to create new applications with MCP
- how to get code running on the MCP card
- debugging the program ■

MCP lets you develop an application on the Macintosh II computer that communicates with processes on the Macintosh II main logic board, tasks on the MCP card or other smart cards, or processes and tasks on both.

During software development, you need to create the following:

- an MCP card application containing an A/ROSE task that will be downloaded to the smart card
- a Macintosh application to run on the Macintosh main logic board that incorporates A/ROSE Prep, the driver that interacts with the A/ROSE task on the smart card program

Before you start

Before learning how to create MCP applications, you should have an understanding of the client/server relationship. (Refer to Chapter 3 for more information.)

The resources and tools you need to develop applications are included on the MCP distribution disks and described in this chapter. You should already have copied all the files provided on the MCP distribution disks to a new folder on your hard disk; if not, do so now by following the instructions in Chapter 3.

Within the MCP folder you created, you should now create another folder for the application you will be working on. You will use the following files to build an MCP card application then download it to the MCP card:

- `A/ROSE:Examples:osmain.c`
- `A/ROSE:Examples:makefile`

Copy these files, then rename them as appropriate for the application you want to build.

△ **Important** To speed development, you should read and use the include files provided on the distribution disk. You should also read and understand the code provided in the Examples files for A/ROSE and A/ROSE Prep. △

The examples in this chapter demonstrate how to build an A/ROSE MCP card application and download it to an MCP card. Development is similar for building a Macintosh application using A/ROSE Prep that runs on the Macintosh II main logic board.

Development is intended to be carried out under MPW, using Assembler and C; the examples in this chapter are written in C. Compile and link your code for A/ROSE as though it were a normal Macintosh application.

You should avoid normal Macintosh run-time libraries; the Macintosh toolbox is not supported by A/ROSE.

How to create an application

In order to use MCP to to create applications that run on a smart card, you will need to:

- create original code for the functions you want an application program to perform (You can use one of the example programs provided on the MCP distribution disks as a starting point for writing your new code, if you prefer.)
- modify the main program (`osmain.c`) by removing any existing code for functions that you do not want (such as the sample tasks currently included) and adding the application program containing your new code
- modify the makefile to compile and link the edited code and the new code for your task(s) with the appropriate A/ROSE library routines

Makefiles are supplied as examples to illustrate the creation of applications for both A/ROSE and A/ROSE Prep. In the examples of code provided in this chapter, any characters highlighted in bold show a change to the code (either added, deleted, or in some way modified).

◆ *Note:* If you want to download dynamically (A/ROSE already present on the card), then you don't have to modify `osmain.c`. For more information, refer to the section on *Generic A/ROSE Downloading* later in this chapter.

Creating new code

You will need to create new code for the functions you want the program to perform. For purposes of this example, the following sample code was created under MPW for a new task to run on an MCP card. This task illustrates how to display message text; this text can also be printed using standard MPW C print procedures.

```
/****************************************************************************/
/*                                                                          */
/*                       example NewTask - A/ROSE                           */
/*                                                                          */
/****************************************************************************/


#include      "os.h"
New_Task  ()
{
        short  i;

        for (i = 0; i < 10; i++)   /* or it could be 100 or 1000!   */
        {
                printf("My TID = %x, Times through the loop = %d, I am here?\n",
                        GetTID(), i);
        }
}
/****************************************************************************/
```

Modifying the main program

The main program initiates both the tasks and MCP software (including A/ROSE and supporting software services).

The file `:A/ROSE:Example:osmain.c` provides a main program written in C as well as examples of tasks. These examples are typical of the highest level of an application that runs on a smart card. The purpose of `osmain` is entirely that of initialization. To initialize A/ROSE, define and start a number of tasks, set the clock rate, and then pass control to A/ROSE.

The main program you create should consist of:

- a call to `osinit()` to initialize A/ROSE. Your code must make this call first, so that the initialization required for the rest of `osmain` can be done.
- a call to `StartTask()` for each developer-created task that is desired
- a call to `StartTask()` for each A/ROSE manager task desired
- any other initialization that needs to be done. This initialization may be hardware dependent or simply appropriate to your application code, such as calling a function to reset the SCC chips after you call `osinit` on the AST card.
- finally, a call to `osstart()` to start the operating system and the tasks

◆ *Note:* After the call to `osstart()`, control is never returned.

StartTask() can also be executed by a running task within A/ROSE once A/ROSE is started. The RSM Manager is specifically designed to support downloading and starting of tasks dynamically.

You should have already created a new main program file by copying the `osmain` file from the folder :A/ROSE:Examples; you are now ready to begin editing that file. Modify this new file to use what you need, delete the example tasks you do not need for your program, and insert code for your own tasks.

For this example, the code for the `osmain` file is highlighted in bold to show some tasks that can be deleted.

```
/*************************************************************************/
/*                                                                       */
/*                        example os main - A/ROSE              .        */
/*                                                                       */
/*                                                                       */
/*     Copyright © 1987,1988 Apple Computer, Inc. All rights reserved.   */
/*                                                                       */
/*************************************************************************/

#include "os.h"
#include "managers.h"
#include "mrdos.h"
#include "siop.h"

void   osinit ();
void   osstart ();
void   name_server ();
void   sccreset  ();
void   time_manager   ();
void   time_tester   ();
void   timeit  ();
void   echo_manager ();
void   echo_example   ();
void   trace_manager ();

#ifdef  PRINT
void  print_manager   ();
#define  PRINT_SLOT  0x0d             /* default slot for printing */
#endif

void  tester  ();
void   ICCM ();
void   remote_manager ();
void  MMSVP  ();
void  MMSVPClient   ();

      pascal void illegal ()
            extern             0x4afc;
main ()
{
      struct ST_PB stpb, *pb;

      unsigned short clock_parms, *cp_ptr;

      osinit (cMaxMsg, cOSStack);
                  /* Init OS with cMaxMsg messages and cStackOS stack   */

      pb = &stpb;

      if  (GetCard  ()  ==  PRINT_SLOT)
            sccreset  ();           /* Be sure SCC is reset...   */
```

```
        /*      Start name server - priority 31, 4k stack, 0 heap.          */

        pb      ->      CodeSegment = NULL;
        pb      ->      DataSegment = NULL;
        pb      ->      StartParmSegment = NULL;
        pb      ->      stack = 4096;
        pb      ->      heap = 0;
        pb      ->      priority = 31;
        pb      ->      InitRegs.PC = name_server;
        pb      ->      InitRegs.A_Registers [5] = GetgCommon() ->      gInitA5;
        pb      ->      ParentTID = GetTID ();

        if (StartTask (pb) == 0)
                illegal ();

#ifdef  PRINT
        if  (GetCard  ()  ==  PRINT_SLOT)
        {
        /*      Start  print  manager  -  priority  30,  4k  stack,  0  heap.
        */

        pb  ->  CodeSegment  =  NULL;
        pb  ->  DataSegment  =  NULL;
        pb  ->  StartParmSegment  =  GetMem  (1);

        /*  Set  print  manager  to  print  from  slot  PRINT_SLOT.  This  allows
*/
        /*  all  cards  to  send  their  output  to  one  slot  for  printing.  If
*/
        /*  printing  is  desired  on  each  card  individually,  then  replace
           */
        /*  the  line  below  with  the  following:
*/
        /*      *(pb  ->  StartParmSegment)  =  GetCard  ();
*/


        *(pb    ->      StartParmSegment)    =    PRINT_SLOT;
        pb      ->      stack  =  4096;
        pb      ->      heap  =  0;
        pb      ->      priority  =  31;
        pb      ->      InitRegs.PC  =  print_manager;
        pb      ->      InitRegs.A_Registers   [5]   =   GetgCommon() ->
        gInitA5;
        pb      ->      ParentTID  =  GetTID();

        if  (startTask  (pb)  ==  0)
                illegal  ();
        }

#endif  PRINT
```

```
/*      Start timer manager - priority 30, 4k stack, 0 heap.
*/

pb      ->      CodeSegment = NULL;
pb      ->      DataSegment = NULL;
pb      ->      StartParmSegment = NULL;
pb      ->      stack = 4096;
pb      ->      heap = 0;
pb      ->      priority = 31;
pb      ->      InitRegs.PC = time_manager;
pb      ->      InitRegs.A_Registers [5] = GetgCommon() -> gInitA5;
pb      ->      ParentTID = GetTID ();

if (StartTask (pb) == 0)
        illegal ();

/*      Start ICC manager - priority 31, 128-byte stack, 0 heap.      */

pb      ->      CodeSegment = NULL;
pb      ->      DataSegment = NULL;
pb      ->      StartParmSegment = NULL;
pb      ->      stack = 128;
pb      ->      heap = 0;
pb      ->      priority = 31;
pb      ->      InitRegs.PC = ICCM;
pb      ->      InitRegs.A_Registers [5] = GetgCommon() -> gInitA5;
pb      ->      ParentTID = GetTID ();

if (StartTask (pb) == 0)
        illegal ();

/*      Start RSM manager - priority 30, 4k-byte stack, 0 heap.       */

pb      ->      CodeSegment = NULL;
pb      ->      DataSegment = NULL;
pb      ->      StartParmSegment = NULL;
pb      ->      stack = 4096;
pb      ->      heap = 0;
pb      ->      priority = 30;
pb      ->      InitRegs.PC = remote_manager;
pb      ->      InitRegs.A_Registers [5] = GetgCommon() ->    gInitA5;
pb      ->      ParentTID = GetTID ();

if (StartTask (pb) == 0)
        illegal ();
/*      Start echo manager - priority 30, 128 stack, 0 heap.          */

pb      ->      CodeSegment = NULL;
pb      ->      DataSegment = NULL;
pb      ->      StartParmSegment = NULL;
pb      ->      stack = 128;
pb      ->      heap = 0;
pb     ->     priority = 30;
pb     ->     InitRegs.PC = echo_manager;
pb     ->     InitRegs.A_Registers [5] = GetgCommon() ->     gInitA5;
pb     ->     ParentTID = GetTID ();

if (StartTask (pb) == 0)
       illegal ();

/*     Start trace manager - priority 30, 1k stack, 0 heap.        */

pb     ->     CodeSegment = NULL;
pb     ->     DataSegment = NULL;
pb     ->     StartParmSegment = NULL;
pb     ->     stack = 1024;
pb     ->     heap = 0;
pb     ->     priority = 30;
pb     ->     InitRegs.PC = trace_manager;
pb     ->     InitRegs.A_Registers [5] = GetgCommon() ->     gInitA5;
pb     ->     ParentTID = GetTID ();

if (StartTask (pb) == 0)
       illegal ();

/*     Start echo example - priority 30, 128 stack, 0 heap.
*/

pb     ->     CodeSegment = NULL;
pb     ->     DataSegment = NULL;
pb     ->     StartParmSegment = NULL;
pb     ->     stack = 128;
pb     ->     heap = 0;
pb     ->     priority = 30;
pb     ->     InitRegs.PC = echo_example;
pb     ->     InitRegs.A_Registers [5] = GetgCommon() -> gInitA5;
pb     ->     ParentTID = GetTID ();

if (StartTask (pb) == 0)
       illegal ();
/*     Start name tester - priority 10, 4k stack, 1024 heap.
*/

pb     ->     CodeSegment = NULL;
pb     ->     DataSegment = NULL;
pb     ->     StartParmSegment = NULL;
pb     ->     stack = 4096;
pb     ->     heap = 1024;
pb     ->     priority = 10;
pb     ->     InitRegs.PC = tester;
pb     ->     InitRegs.A_Registers [5] = GetgCommon() -> gInitA5;
pb     ->     ParentTID = GetTID ();

if (StartTask (pb) == 0)
```

```
                illegal ();

        /*      Start timer tester - priority 10, 4k stack, 0 heap.
        */

        pb      ->      CodeSegment = NULL;
        pb      ->      DataSegment = NULL;
        pb      ->      StartParmSegment = NULL;
        pb      ->      stack = 4096;
        pb      ->      heap = 0;
        pb      ->      priority = 10;
        pb      ->      InitRegs.PC = time_tester;
        pb      ->      InitRegs.A_Registers [5] = GetgCommon() -> gInitA5;
        pb      ->      ParentTID = GetTID ();

        if (StartTask (pb) == 0)
                illegal ();

        /*      Start timerit - priority 10, 4k stack, 0 heap.      */

        pb      ->      CodeSegment = NULL;
        pb      ->      DataSegment = NULL;
        pb      ->      StartParmSegment = NULL;
        pb      ->      stack = 4096;
        pb      ->      heap = 0;
        pb      ->      priority = 10;
        pb      ->      InitRegs.PC = timeit;
        pb      ->      InitRegs.A_Registers [5] = GetgCommon() -> gInitA5;
        pb      ->      ParentTID = GetTID ();

        if (StartTask (pb) == 0)
                illegal ();

        /*      WHJW: Start MMSVP - priority 10, 4k stack, 0 heap. */
        /*      This is provided for diagnostic purposes.          */

        pb      ->      CodeSegment = NULL;
        pb      ->      DataSegment = NULL;
        pb      ->      StartParmSegment = NULL;
        pb      ->      stack = 4096;
        pb      ->      heap = 0;
        pb      ->      priority = 10;
        pb      ->      InitRegs.PC = MMSVP;
        pb      ->      InitRegs.A_Registers [5] = GetgCommon() -> gInitA5;
        pb      ->      ParentTID = GetTID ();

        if (StartTask (pb) == 0)
                illegal ();

        /*      WHJW: Start MMSVP client task - priority 11, 4k stack, */
                        /*      0 heap. This is provided for diagnostic
purposes.*/
```

```
            pb      ->      CodeSegment   =  NULL;
            pb      ->      DataSegment   =  NULL;
            pb      ->      StartParmSegment   =  NULL;
            pb      ->      stack  =  4096;
            pb      ->      heap  =  0;
            pb      ->      priority  =  11;
            pb      ->      InitRegs.PC   =   MMSVPClient;
            pb      ->      InitRegs.A_Registers   [5]   =   GetgCommon()   ->  gInitA5;
            pb      ->      ParentTID  =  GetTID  ();

            if  (StartTask  (pb)  ==  0)
                  illegal  ();

            /*     Start  operating  system.                                 */


      #ifdef  AST_ICP
            /*  setup  VIA  to  interrupt  us  every  10  milliseconds        */
            clock_parms   =   VIA_TICK_RATE;        /*  clock  rate  for  10  ms  tick
      */
            cp_ptr   =   &clock_parms;
      #endif   AST_ICP

      #ifdef  MCP
            cp_ptr  =  NULL;
      #endif  MCP

            osstart        (TICK_MIN_MAJ,  TICKS_PS,  cp_ptr);       /*  start  things  up
            */
            illegal  ();                        /*  should  never  get  here          */
      }

      /***************************************************************************/
```

Next, edit the file to remove the tasks highlighted, and then insert code for the new task (named `NewTask`). The main program file for this example should now look like this:

```
/***************************************************************************/
/*                                                                         */
/*                    example os main - A/ROSE                             */
/*                                                                         */
/*                                                                         */
/*    Copyright © 1987,1988, 1989 Apple Computer, Inc. All rights reserved.
      */
/*                                                                         */
/***************************************************************************/

#include "os.h"
#include "managers.h"
#include "mrdos.h"
#include "siop.h"

void   osinit ();
void   osstart ();
void   name_server ();
void   echo_manager ();
void   trace_manager ();
void   ICCM ();
void   remote_manager ();
void   New_Task   ();

pascal void illegal () extern     0x4afc;

main ()
{
       struct ST_PB stpb, *pb;

       unsigned short clock_parms, *cp_ptr;

       osinit (cMaxMsg, cOSStack);
                    /* Init OS with cMaxMsg messages and cStackOS stack   */

       pb = &stpb;

       /*     Start name server - priority 31, 4k stack, 0 heap.          */

       pb     ->     CodeSegment = NULL;
       pb     ->     DataSegment = NULL;
       pb     ->     StartParmSegment = NULL;
       pb     ->     stack = 4096;
       pb     ->     heap = 0;
       pb     ->     priority = 31;
       pb     ->     InitRegs.PC = name_server;
       pb     ->     InitRegs.A_Registers [5] = GetgCommon() ->     gInitA5;
       pb     ->     ParentTID = GetTID ();
```

```
if (StartTask (pb) == 0)
        illegal ();

/*      Start ICC manager - priority 31, 128-byte stack, 0 heap.     */

pb      ->      CodeSegment = NULL;
pb      ->      DataSegment = NULL;
pb      ->      StartParmSegment = NULL;
pb      ->      stack = 128;
pb      ->      heap = 0;
pb      ->      priority = 31;
pb      ->      InitRegs.PC = ICCM;
pb      ->      InitRegs.A_Registers [5] = GetgCommon() ->     gInitA5;
pb      ->      ParentTID = GetTID ();

if (StartTask (pb) == 0)
        illegal ();

/*      Start RSM manager - priority 30, 4k-byte stack, 0 heap.      */

pb      ->      CodeSegment = NULL;
pb      ->      DataSegment = NULL;
pb      ->      StartParmSegment = NULL;
pb      ->      stack = 4096;
pb      ->      heap = 0;
pb      ->      priority = 30;
pb      ->      InitRegs.PC = remote_manager;
pb      ->      InitRegs.A_Registers [5] = GetgCommon() ->     gInitA5;
pb      ->      ParentTID = GetTID ();

if (StartTask (pb) == 0)
        illegal ();

/*      Start echo manager - priority 30, 128 stack, 0 heap.         */

pb      ->      CodeSegment = NULL;
pb      ->      DataSegment = NULL;
pb      ->      StartParmSegment = NULL;
pb      ->      stack = 128;
pb      ->      heap = 0;
pb      ->      priority = 30;
pb      ->      InitRegs.PC = echo_manager;
pb      ->      InitRegs.A_Registers [5] = GetgCommon() ->     gInitA5;
pb      ->      ParentTID = GetTID ();

if (StartTask (pb) == 0)
        illegal ();

/*      Start trace manager - priority 30, 1k stack, 0 heap.         */

pb      ->      CodeSegment = NULL;
pb      ->      DataSegment = NULL;
```

```
       pb     ->     StartParmSegment = NULL;
       pb     ->     stack = 1024;
       pb     ->     heap = 0;
       pb     ->     priority = 30;
       pb     ->     InitRegs.PC = trace_manager;
       pb     ->     InitRegs.A_Registers [5] = GetgCommon() ->     gInitA5;
       pb     ->     ParentTID = GetTID ();

       if (StartTask (pb) == 0)
              illegal ();

       /*     Start New_Task - priority 20, 4k stack, 0 heap.                  */

       pb     ->     CodeSegment = NULL;
       pb     ->     DataSegment = NULL;
       pb     ->     StartParmSegment = NULL;
       pb     ->     stack = 4096;
       pb     ->     heap = 0;
       pb     ->     priority = 20;
       pb     ->     InitRegs.PC = New_Task;
       pb     ->     InitRegs.A_Registers [5] = GetgCommon() -> gInitA5;
       pb     ->     ParentTID = GetTID ();

       if (StartTask (pb) == 0)
              illegal ();

       /*     Start operating system.                                          */

       cp_ptr = NULL;

       osstart      (TICK_MIN_MAJ, TICKS_PS, cp_ptr);/* start things up       */
       illegal ();                      /* should never get here               */

}

/****************************************************************************/
```

Modifying the makefile

Now that you have modified `osmain` to include the code for your new task, next you will modify the makefile. You should have already copied the makefile from the folder :A/ROSE:Examples; if so, you are now ready to modify that new file. Using the makefile, you can do the following:

- compile the initialization software (`osmain.c`) and application tasks
- link the desired A/ROSE libraries with the application tasks and initialization software to build the program to be downloaded to the smart card

Compile and link this code as though it were a normal Macintosh application. You should not use normal Macintosh run-time libraries; the A/ROSE operating system does not support the Macintosh toolbox. The next two sections direct you to the available include files and libraries you can use to modify the makefile.

A/ROSE include files

Table 8-1 lists some of the include files available and briefly describes each file. These include files are located in the folder `A/ROSE:includes:` on the MCP distribution disks. You can use these include files to compile and link your code.

- **Table 8-1** Include files

Assembly Filename	C Language Filename	Description of File
`os.a`	`os.h`	Defines the operating-system message structure, commonly used constants, and externally visible system library routines.
`managers.a`	`managers.h`	Contains the structures and constants used when accessing the Name Manager, Time Manager, and InterCard Communications Manager.
`arose.a`	`arose.h`	Contains constants and structures for the operating-system tables.

In addition, there are four include files similar to those listed in Table 8-1 specifically for use with the AST-ICP card; these files are named `scc.a, scc.h, siop.a, and siop.h` and are located in the A/ROSE:includes: folder on the distribution disks. These files are useful if any SCC hardware is to be used.

A/ROSE libraries

The file `:A/ROSE:Binaries:os.o` is the library containing the generic A/ROSE routines. The file `:A/ROSE:Binaries:osglue.o` is the glue (interface) library containing code to allow tasks to use A/ROSE utility routines.

◆ *Note:* Do not use the standard C library `cruntime.o`; the `osglue.o` file that is provided on the MCP distribution disks contains run-time library routines.

When you're ready, use the MPW Link command to link your code with these files .

△ **Important** To avoid conflicts in the MPW linker with duplicate names, you should prefix all nonvisible and externally invisible C function and subroutine names with static. Doing this reduces the possibility that routines with the same names from different object files will interact to produce linker errors. △

Changes to the makefile

The following code from the new file (the sample file that you copied) is highlighted in **bold** to show the tasks that changed or were deleted from the makefile. Compare this file with the one following to determine the code that has been changed, added, or deleted.

◆ *Note:* {`card`} represents a string that you will replace.

```
/***********************************************************************
****/
/*
        */
/*                           Makefile for example download.
        */
/*
        */
/*
        */
/*      Copyright © 1987, 1988  Apple Computer, Inc. All rights
reserved. */
/*
        */
/***********************************************************************
*****/


#       Makefile for test download.

#       To invoke this makefile please type
#       make



Card            =       Binaries



CI                      =       ::includes:

LinkOpts        =       -l -x :"{Card}":start.xrf > :"{Card}":start.map



AOptions        =       -d &Card=∂'{Card}∂' -i ::includes:,::"{Card}": -l -
font Courier,7 ∂

        -pagesize 115,124 -print Data,Obj,Lits,NoMDir

COptions        =       -D "{Card}" -D PRINT -i {CI}

CSources        =       ::includes:scc.h echo.c trace_manager.c
pr_manager.c printf.c name_tester.c ∂

                                timer_tester.c osmain.c timeIt.c L3osmain.c
L3MMSVP.c L3MMSVPClient.c rs.c

AsmLists        =       ossccint.a.lst IOPNub.a.lst L3MMSVP.a.lst

Targets                 =       ≈.o ≈.lst start GenAROSE
```

```
:"{Card}":    ƒ       : ::includes:


all                 ƒ      :"{Card}":start      :"{Card}":GenAROSE


:"{Card}":start     ƒ      :"{Card}":osmain.c.o  ::"{Card}":OS.o
::"{Card}":osglue.o ∂

              :"{Card}":printf.c.o  :"{Card}":name_tester.c.o
:"{Card}":timer_tester.c.o ∂

              :"{Card}":timeIt.c.o  :"{Card}":echo.c.o
:"{Card}":trace_manager.c.o ∂

               :"{Card}":ossccint.a.o  :"{Card}":pr_manager.c.o

       link -t 'DMRP' -c 'RWM ' -o :"{Card}":start ∂

              :"{Card}":osmain.c.o  :"{Card}":ossccint.a.o
::"{Card}":OS.o  ::"{Card}":osglue.o ∂

              :"{Card}":pr_manager.c.o  :"{Card}":printf.c.o
:"{Card}":name_tester.c.o  :"{Card}":timer_tester.c.o ∂

              :"{Card}":timeIt.c.o  :"{Card}":echo.c.o
:"{Card}":trace_manager.c.o ∂

              {LinkOpts}


:"{Card}":GenAROSE  ƒ      :"{Card}":GenAROSE.c.o  ::"{Card}":OS.o
::"{Card}":osglue.o

       link -t 'DMRP' -c 'RWM ' -o :"{Card}":GenAROSE ∂

              :"{Card}":GenAROSE.c.o  ::"{Card}":OS.o  ::"{Card}":osglue.o
{LinkOpts} -l >GenAROSE.map


:"{Card}":osmain.c.o       ƒ    {CI}os.h  {CI}managers.h  {CI}arose.h
{CI}siop.h
```

```
:"{Card}":GenAROSE.c.o     ƒ    {CI}os.h {CI}managers.h {CI}arose.h


:"{Card}":ossccint.a.o     ƒ      ::"{Card}":OSDefs.d


:"{Card}":pr_manager.c.o   ƒ      {CI}scc.h {CI}siop.h {CI}os.h
{CI}managers.h


:"{Card}":printf.c.o       ƒ      {CI}os.h {CI}managers.h


:"{Card}":echo.c.o         ƒ      {CI}os.h


:"{Card}":trace_manager.c.o        ƒ      {CI}scc.h {CI}siop.h {CI}os.h
{CI}managers.h


:"{Card}":name_tester.c.o  ƒ      {CI}os.h {CI}managers.h {CI}arose.h


:"{Card}":timer_tester.c.o         ƒ      {CI}os.h {CI}managers.h


:"{Card}":timeIt.c.o               ƒ      {CI}os.h {CI}managers.h


:"{Card}":L3MMSVP.c.o              ƒ      {CI}os.h {CI}diags.h


:"{Card}":L3MMSVP.a.o              ƒ      {CI}arose.a {CI}os.a
{CI}diags.a {CI}siop.a


:"{Card}":L3MMSVPClient.c.o        ƒ      {CI}os.h {CI}diags.h
{CI}managers.h
```

```
#       Special targets.


#       Listings - Print changed files.


Listings     ƒƒ     {AsmLists}

        Print -f Courier -s 7 -ls 0.70 -r {NewerDeps}


Listings     ƒƒ     {CSources}

        Print -f Courier -s 7 -ls 0.70 -r -hf Courier -hs 9 -h -n
{NewerDeps}

        echo "Last listings made `Date`." > Listings


#       Clean - Remove all targets.


Clean   ƒ      {Targets}

        Delete -i {Targets}
```

Compiling and linking your code

You will next use the makefile to generate the commands that will compile and link your code together. To do so, enter the MPW command `Make`.

The commands produced are:

```
C -D "Binaries" -D PRINT -i "::MCP Software:A/ROSE:"includes: osmain.c -o
osmain.c.o
C -D "Binaries" -D PRINT -i "::MCP Software:A/ROSE:"includes: NewTask.c ∂
        -o newTask.c.o
Link -t 'GMSC' -c '????' -o start ∂
        osmain.c.o "::MCP Software:A/ROSE:MCP:"OS.o ∂
        "::MCP Software:A/ROSE:MCP:"osglue.o ∂
        "::MCP Software:A/ROSE:Examples:MCP:"printf.c.o ∂
        "::MCP Software:A/ROSE:Examples:MCP:"trace_manager.c.o ∂
        NewTask.c.o -l -x xref > map
```

◆ *Note:* `{AROSE}` is the pathname of the A/ROSE folder under MPW. You must set this up when using MPW; otherwise, you must substitute the full pathname for `{AROSE}`.

Table 8-2 defines the parameters to the `Link` command, shown in the example above.

■ **Table 8-2** Link command parameters

Parameter	**Description**
`-t`	The type of file that `Link` command is going to generate
`GMSC`	The file type convention GMSC takes is for the MCP card but the file type can be anything.
`-c`	The creator
`'????'`	Enter any appropriate creator name
`-o start`	The output file from the linker; the file `start` will be created in your directory
`osmain.c.o`	The initialization routine that you modified
`os.o`	File that contains A/ROSE operating system
`osglue.o`	File containing glue code
`printf.c.o`	Printing subroutine source code for A/ROSE; equivalent to the `printf` routine in standard C
`trace_manager.c.o`	Tracing tool for A/ROSE
`NewTask.c.o`	The name of the main program containing your task

◆ *Note* Only the globally-visible name of the task should be the task's main program. The task's main routine should not be called "main" but must be given another name, because your code is sharing space with the entire operating system, and the name `osmain` is always visible.

Select the entire section listed above to enter and execute these commands; this creates the application that you will download to an MCP card.

Downloading code to the MCP card

Download is an MPW tool that downloads smart card application files to smart cards. . The makefile in :A/ROSE:Examples produces the following one executable file for downloading:

```
:A/ROSE:Examples:Binaries:start
```

This section first discusses the Download tool, then presents information to help you create your own download application.

Generic A/ROSE downloading

You can download a generic version A/ROSE with a single call. With this feature you can download an executable ready to use A/ROSE onto a NuBus card without the effort of modifying "`osmain.c`", linking with A/ROSE routines, and downloading the initial module. `StartAROSE` will halt the card, download A/ROSE module with commonly used managers listed below, and start the card. The generic version of A/ROSE is called, `StartAROSE`. The `StartAROSE` routine is defined in download_lib.o

Managers included in the generic version of A/ROSE are:

- Name Manager
- InterCard Communications Manager
- Remote System Manager
- Echo Manager

The generic A/ROSE module is placed in the A/ROSE Prep file under the resource 'DMRP". The call to `StartAROSE` makes a call to `NewDownload` to download this resource from the A/ROSE Prep file in the system folder.

This is the Calling Sequence:

```
tid_type StartAROSE(short slotNum);
              /*  slotNum is value between 0x9 and 0xE  */
```

`startAROSE` returns the Task ID of the Name Manager. Generic A/ROSE is part of the A/ROSE Prep file. This facility enables Apple to update the OS. You can download the generic A/ROSE and your other files using the dynamic download facility. In a later release you will be able to specify the managers to be included in the Generic A/ROSE.

The Return Status will indicate if the card could not be downloaded for any reason by returning a value of 0.

Calling the Download tool

The name of the file to be downloaded and the destination slot number or numbers are provided as parameters. The calling sequence for the Download tool is

`Download Filename [-S1 ... -Sn]` where: `Filename` specifies the name of the program file to be downloaded to the card, and `sn` is the slot where the card is found.

Slots are numbered in hex from 9 to E (left to right); two examples might be -9 or -A. You can specify multiple slots. If you do not specify a slot number, the default for Download is all slots containing smart cards of the kind matching the Download tool.

After validating these parameters, Download does the following:

- performs the download for each of the slots selected
- copies the resources of the object file (including Jump Table, Data Initialization, and Segments) into RAM of the selected smart cards
- starts each card when Download sets the program counter to the appropriate address

You can now download the compiled and linked code to the smart card for execution, using the Download tool provided on the MCP distribution disk.

To continue the example from the makefile presented earlier in this chapter, follow the steps described next. To download the sample application to the card, enter

```
"::MCP Software:A/ROSE:Downloader:Download" start
```

Next, enter the following comand:

```
directory "::MCP Software:A/ROSE Prep:Examples:"
pr_manager
```

This command starts up the MPW Print Manager tool. Using this tool, you can check if the downloaded card is running and able to send messages to tasks running on the Macintosh II, and then display results on the screen (similar to the example shown next).

```
Print Manager TID = 4
Starting Main Loop
TID b05: Trace Manager: Starting.
My TID = b06, Times through the loop = 0, I am here
My TID = b06, Times through the loop = 1, I am here
My TID = b06, Times through the loop = 2, I am here
My TID = b06, Times through the loop = 3, I am here
My TID = b06, Times through the loop = 4, I am here
My TID = b06, Times through the loop = 5, I am here
My TID = b06, Times through the loop = 6, I am here
My TID = b06, Times through the loop = 7, I am here
My TID = b06, Times through the loop = 8, I am here
My TID = b06, Times through the loop = 9, I am here
My TID = b06, Times through the loop = 10, I am here
```

To stop using the MPW print manager tool, press Command-period; the screen displays

```
CloseQueue Called.
```

Download errors

Download errors are indicated by messages to the `stderr` file. The state of any cards to be downloaded is undefined if an error is returned. `DLE_NOERR` is a normal return. *Table 8-3* lists Download error constants; these constants are found in the file `:A/ROSE:includes:Download.h.`

■ **Table 8-3** Error constants for Download

Error Displayed	Number	Description
`#define DLE_NOERR`	0	No error
`#define DLE_NOJT`	1	No jump table found
`#define DLE_DATAINIT`	2	Bad Data Init segment
`#define DLE_GLOBALF`	3	Global data-format error
`#define DLE_CODES`	4	Code segment error
`#define DLE_MAC2`	5	Code only runs on Macintosh II
`#define DLE_EMPTY`	6	No cards found
`#define DLE_NOCARD`	7	Slot specified is empty
`#define DLE_RESFILE`	8	Couldn't open resource file
`#define DLE_FILEWRONG`	9	Download file is wrong type
`#define DLE_STARTERR`	10	Starting error
`#define DLE_NOMEM`	11	No memory
`#define DLE_RSMERR`	12	RSM error
`#define DLE_NORSM`	13	No RSM
`#define DLE_NOAROSE`	14	No A/ROSE running on card
`#define DLE_NORSRC`	15	No 'CNFG' resource
`#define DLE_NOPREP`	16	No A/ROSE Prep file
`#define DLE_ABORT`	17	Download aborted

Using the download subroutines

You can use two methods to download code onto NuBus cards. Use the first method during start up when there is nothing loaded on the cards. With this method, your code is downloaded along with A/ROSE operating system code. To implement this first method, you must halt the card, download the code and the operating system, and then start the card. Use the second method to download your tasks when the card is already running the A/ROSE operating system. In this case, the card should not be halted or started.

Use the following subroutines to implement the two methods just described for downloading code onto cards:

- NewDownload
- DynamicDownload

NewDownload is a general download subroutine that can be used to download A/ROSE and user programs initially onto a card. `DynamicDownload` is provided specifically to download user tasks dynamically onto cards running A/ROSE. These routines are supplied in an object library module named "download-lib.o," found in the A/ROSE:Downloader folder The object library also includes other useful subroutines such as `TestSlot, NewFindcard, StartCard, and HaltCard.` These subroutines read card specific information from the A/ROSE Prep file. This prep file must be in the System Folder of the startup volume. Various parameters can be passed as arguments to these subroutines to control downloading.

◆ *Note:* The `NewDownload` subroutine neither halts nor starts a card. User programs must issue the desired calls specifically when using the NewDownload subroutine. This is to allow users the option to download code and set up user-specific data before starting the card.

These two subroutines support important features that can specify the following:

- load address (where the downloaded program goes),
- the address of the `gCommon` area,
- the resource type of the code (default is 'CODE',)
- the address offset to be used when accessing the card memory,
- the start parameter segment address, and the length of the start parameter segment.
  The parameters that are specifically passed depend on the subroutine you are calling. For example, operations such as initializing the common area can be specified using the options parameter. In any case, your code can be downloaded dynamically, when the card is running A/ROSE, by setting a bit in the options parameter. You can specify the start parameter segment when using `DynamicDownload`.

Although `NewDownload` can be used to download tasks dynamically, it is provided specifically for the initial download of the A/ROSE system and manager tasks. You must specifically *halt* the card before downloading and *start* the card after downloading. You must not call start/halt card for `DynamicDownload`.

Your installation disk contains a sample Macintosh application that can be used to download your code and A/ROSE onto NuBus cards. The source code for this application, `NDLD.c`, is also provided. `NDLD.c` can be used as an example of setting up parameters for the `NewDownload` subroutine.

◆ *Note:* The Download subroutine and MPW tool `download` released with an earlier version of A/ROSE will not work with this version of A/ROSE. In addition, the resource file that contains the code to be downloaded must be the current resource file when you call download. A download tool is provided in the Downloader folder.

To use any of the download subroutines your code must be linked with `download-lib.o`. Your programs (only code running on the main Macintosh logic board) can then call the download subroutines to download code to NuBus cards. The following sections describe the calling mechanism for both the download subroutines and the `ndld` sample application.

NewDownload

To use the `NewDownload` subroutine you must have the A/ROSE Prep file in the system folder. The format of the A/ROSE Prep file will be found in a future Macintosh tech note.

The following shows the NewDownload format

```
#Include "Download.H"

typedef pascal void (*PascalPtrLong)(long segSize);

pascal short NewDownload(slotNUM, addrOffset, loadaddr,

             gCommonAddr, options, restype, registers, ProgProc)

short slotNUM;

/* Slot number to use when loading code. This parameter is NOT a bit
mask. Specify a value between 0x9 and 0xE.  */


long  addrOffset;

/* Offset from address of card to use as beginning. The default
address is defined by the symbol DEF_ADDROFFSET in file Download.h in
the A/ROSE includes folder */


long  loadaddr;

/* Address RELATIVE to addrOffset on the card to load data and code.
The default initial load address of A/ROSE is defined by the symbol
DEF_LOADADDR in file download.h in the A/ROSE includes folder.  */


struct gCommon *gCommonAddr;

/* Address RELATIVE  to addrOffset on the card to load locate the gCommon area.  The  default initial load
address of A/ROSE gCommon is defined by the symbol DEF_GCOMMON in file Download.h in the A/ROSE
includes folder. */


short  options;

/* Set to DL_INITLOAD | DL_CLEARMEM if an initial download and low
memory is to be cleared. Set to DL_INITLOAD only if an initial download
and low memory is not to be cleared.  Set to 0 if a dynamic download.
(Low memory will not be cleared.) If an initial download is being done
then low memory, gCommon, and the jump tables will be set up.  If a
dynamic download is being done then low memory will NOT be touched.  */


ResType  restype;
```

```
/* The resource type (eg. 'CODE') of the resource to load into the
card.  */
```

```
struct ST_Registers *registers;

/* Pointer to a register area (defined in os.h in the A/ROSE includes
folder) where the correct registers will be returned for use in a
RSM_StartTask request to start a task loaded dynamically.  Users need
to specify only an address of an area of structure ST_Registers */
```

```
PascalPtrLong ProgProc;

/*  Progress report procedure.  Called with length of code being
downloaded  */
```

You can create a progress report procedure to monitor the download process continuously if you are downloading a large program. The progress report procedure is declared as a Pascal procedure that takes one long word as a parameter. Whenever the downloader is about to download a segment it calls the progress procedure with the size of the segment being downloaded.

◆ *Note:* `addrOffset` must be 0 and `gCommonAddr` must be 0x400.

`loadaddr` must be greater than or equal to 0x800.

Return status

The `NewDownload` subroutine can return any of the following error constants. The state of the card to be downloaded is undefined if an error is returned. DLE_NOERR is a normal, successful return. The following error constants are taken from the include file `Download.h` located in the folder: `A/ROSE:includes:`

```
/* Error Constants */

#define  DLE_NOERR      0     /* No error                  */

#define  DLE_NOJT       1     /* No jump table found       */

#define  DLE_DATAINIT   2     /* Bad Data Init segment     */

#define  DLE_GLOBALF    3     /* Global data format error  */
```

```
#define  DLE_CODES      4    /* Code segment error            */
#define  DLE_MAC2       5    /* Can only run on Mac II family */
#define  DLE_EMPTY      6    /* Slot specified is empty       */
#define  DLE_NOCARD     7    /* No cards found                */
#define  DLE_STARTERR   10   /* Starting error                */
#define  DLE_NOMEM      11   /*  No memory                    */
#define  DLE_RSMERR     12   /* RSM error                     */
#define  DLE_NORSM      13   /* No RSM                        */
#define  DLE_NOAROSE    14   /* No A/ROSE running on card     */
#define  DLE_NORSRC     15   /* No 'CNFG' resource            */
#define  DLE_NOPREP     16   /* No A/ROSE prep file           */
#define  DLE_ABORT      17   /* Aborting download             */
```

▲ **Caution** The resource file must be open before the call to Download and must be the top most resource. The resource type of the resource to be downloaded is specified in restype. It is the caller's responsibility to call `HaltCard()` and `StartCard()` to halt and start the hardware if it is an initial load. Supporting routines described in the section titled Supporting Routines have been provided to assist you. ▲

DynamicDownload

To use DynamicDownload you must have the A/ROSE Prep file in the system folder and A/ROSE must be active and running on the NuBus card plugged in the specified slot.

```
pascal tid_type DynamicDownload(slotNUM, restype,
       st_parmblock, startParmSegment, lenParmSegment)
```

```
short  slotNUM;

/* slot number to use when loading code. This is NOT a bit mask.
Normally a value between 0x9 and 0xE.  */
```

```
ResType  restype;

/* The resource type (eg. 'CODE') of the resource which has the code to
load into the card.  */
```

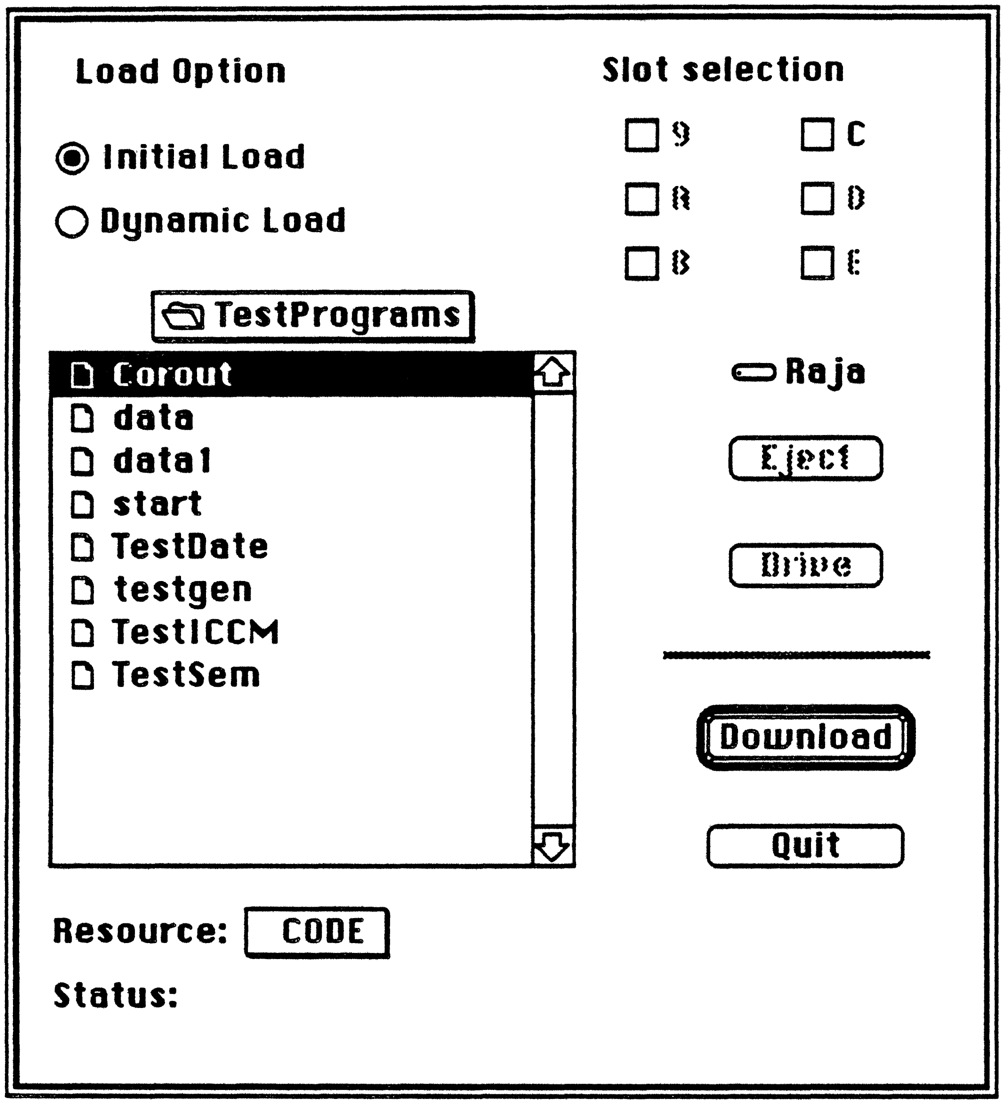
```
struct ST_PB *st_parmblock;

/* Pointer to a RSM_StartTask parameter area (defined in os.h) in the
A/ROSE includes folder). It is the caller's respnsibility of to
initialize the following fields:



st_parmblock->stack

st_parmblock->heap

st_parmblock->priority

st_parmblock->ParentTID



The updated parameter block along with the reply from Remote System
Manager is returned.  */



char *startParmSegment;

/* Starting parameters for the downloaded task.  */



long lenParmSegment;

/* Length of the startParmSegment  */
```

DynamicDownload returns the Task ID of the downloaded task. If the task could not be downloaded, a value of 0 is returned.

NewDownload (NDLD)

NDLD is an application used to download software onto smart NuBus Cards. NDLD calls the NewDownload subroutine. NDLD is launched like any other Macintosh application. When ndld is launched it displays a dialog box listing all the options available. Figure 8-1 shows the NDLD Dialog Box. A translation of the dialog box options follows the illustration.

Figure 8-1 The NDLD Dialog Box



● Initial Load

○ Dynamic Load

Resource: CODE

Status:

Load option Offers button selection for either initial or dynamic load.

Slot selection Specifieds the computer slots into code should be downloaded

Standard Getfile File to be downloaded.

Download Download start button

Quit End download dialog and quit the application

Resource Type of the resource to be downloaded. Default is 'CODE'

Interface specification

NewDownload and DynamicDownload subroutines are designed to be linked with applications running on the Macintosh . The following is the only object library module provided that must be linked to use the Download subroutines:

`Download-lib.o` (located in the A/ROSE:Downloader folder)

For example., the following link statement is used for linking NDLD with downloader.

```
        Link -w -t APPL -c '????' ∂

                Download-lib.o ∂

                "{Libraries}"Interface.o ∂

                "{CLibraries}"CRuntime.o ∂

                "{CLibraries}"StdCLib.o ∂

                "{CLibraries}"Math.o ∂

                "{CLibraries}"CInterface.o ∂

                -o NDLD
```

Load module description

The load module generated by the MPW linker is a relocatable code with jump tables. The code is downloaded to the card by the `NewDownload` subroutine starting at the location you specify(default is 0x800). The `NewDownload` subroutine fixes the addresses of the code in memory and the segment entries in the jump table are set to a loaded <<What does loaded mean?>> state.

In dynamic download, an area large enough to accommodate the task being downloaded and the starting parameter segments is allocated in the card using the Remote System Manager `GetMem` command. The task is downloaded into the card and started using the Remote System Manager `StartTask` command.

Implementation strategy

The A/ROSE Prep file is very important for the downloader to work. Start/Halt routines and the NewDownload subroutine first call the `TestSlot` routine to get information about the card to be downloaded. This information includes the BoardID of the card and the actual address that is used to download from the Macintosh. The code that does the actual download is also in the A/ROSE

Prep file in the resource named 'dwnl'. This allows the download code to be modified in future releases without the need for users to relink their code with a new release of A/ROSE.

A/ROSE Prep file

The A/ROSE Prep file contains information that is used by the download subroutines. Some of this information is specific to the type of card. The A/ROSE Prep file must be in the System Folder before any programs are downloaded onto NuBus cards.

There is a unique Board ID for each type of NuBus card. These board IDs are assigned by Apple. The Slot Manager information on the cards is used to determine the Board ID of each card in the system. The Board ID is used as the resource ID to get card specific information, such as routines to start/halt card.

The format of the A/ROSE Prep file will be described in a later Tech Note.

The following sections describe supporting routines that aid in the maintenance of the card. The specific tasks are discussed as well as how to perform them.

TestSlot

This test determines if a card is capable of running A/ROSE. TestSlot checks a slot for a card with valid configuration information and returns the characteristics of the card. The BoardID is unique for each type of card. The starting address of RAM and the maximum length of RAM are returned in `startMemPtr` and `lenMemPtr`, respectively. The starting address of the RAM area is returned as a NuBus address and the length of the RAM is the maximum amount of RAM that the architecture of the board allows. The download subroutine finds the actual size of the RAM physically present in the card at the time of download. `CPURsrcPtr` and `networkRsrcPtr` are pointers to two arrays of 4 short words.

```
short  TestSlot(slotNUM,  boardIDPtr,  lenMemPtr,  startMemPtr,

                          CPURsrcPtr,  networkRsrcPtr)

short  slotNUM;

/*   0x9 through 0xE.   Not a bit mask   */

short  *boardIDPtr;         /* place where the board ID is returned   */

long  *lenMemPtr,  *startMemPtr;

/*   lenMemPtr is the maximum length of RAM and startMemPtr is the
starting address of the RAM area   */

short  CPURsrcPtr[4],  networkRsrcPtr[4];
```

```
/*  CPURsrcPtr is a double long word area where the CPU resource info
is returned and networkRscrPtr is a double long word where the network
resource info is returned  */
```

`TestSlot` returns zero if the slot does not contain a card capable of running A/ROSE. TestSlot returns -1 for slot zero if the card is capable of running A/ROSE.

The following table shows the Board ID of some existing boards:

Board type	Board IDs
MCP board	10, 11, 13, and 25
ASIC MCP board	24 and 105
AST ICP board	261
Green Spring board	441

NewFindCard

This test helps you find all cards loaded in the system

```
pascal short NewFindcard(slot, boardID)

short *slot, boardID;
```

`*slot` is a bit mask indicating which slots are available for loading. Bit 0 is the least significant bit. Bit9 (bit mask 0x200) corresponds to slot 9. Bit 14 (bit mask 0x4000) corresponds to slot E.

`NewFindcard` will return DLE_NOERR if atleast one card of the correct type is found and will return DLE_EMPTY if no card of the correct type is found.

If the specified boardID is zero, then `NewFindcard` will return notification of all cards capable of running A/ROSE.

If the specified boardID is non zero, then `NewFindcard` will find all the cards that match the boardID. BoardID information is found in the configuration ROM of the card.

StartCard

The start card routine resets the CPU and the NuBus card starts the execution of the downloaded program. In the case of Motorola 68000 processor the program counter is loaded with the value in location 4 and the stack pointer is loaded with the value specified in location 0. A NuBus card should not be started without first downloading code into it.

```
short  StartCard  (SlotNum)

short  SlotNum;              /*   0x9 through 0xE   */
```

where `SlotNum` contains the slot number of the card to start or halt.

HaltCard

The halt card routine halts the execution of programs on the NuBus card by permanently activating the reset line. You must halt the card before doing an initial download.

```
short  HaltCard  (SlotNum)

short  SlotNum;              /*   0x9 through 0xE   */
```

where `SlotNum` contains the slot number of the card to start or halt.

◆ *Note:* The return values for start/halt card routines are same as for download calls.

Example: (Compiled with MPW C 3.0)

To initially load and start a card in Slot 0x0D the NewDownload subroutine would be called in the following way:

```
short  slotNum;

long  addrOffset,  loadaddr;
```

```
struct gCommon *gCommonAddr;
short initial_load;
ResType Resource;
struct ST_Registers Registers;
PascalPtrLong progProc;
short refNum;
.
.
.

slotNum = 0x0d;
addrOffset = 0;
loadaddr = INIT_LOAD;  /* 0x800 */
gCommonAddr = INIT_GCOM;  /* 0x400 */
initial_load = DL_INITLOAD | DL_CLEARMEM;
progProc = NIL;
Resource = 'CODE';

if ((refNum=OpenResFile("\pSampleCode")) != -1)
{
        if (initial_load)
              HaltCard (slotNum);

        status = NewDownload (slotNum, addrOffset, loadaddr,
                      gCommonAddr, initial_load, Resource,
                      &Registers, progProc);
        if (status == DLE_NOERR && initial_load)

              StartCard (SlotNum);


  closeFile (refNum);

}
```

To dynamically download and start a user task in Slot 0x0D, the `DynamicDownload` subroutine would be called in the following way:

```
short slotNum;

struct ST_PB st_parmblock;

ResType Resource;

char startParmSeg[] = {"-c 'ASDF' -t"};

                              /* start parameters for the user task */

long lenParmSeg;

short refNum;

tid_type taskID;

.

.

.


slotNum = 0x0d;

Resource = 'CODE';

lenParmSeg = sizeof startParmSeg;

st_parmblock.stack = 0x1000;

st_parmblock.heap = 0;
st_parmblock.priority = 10;

st_parmblock.ParentTID = 0;


if ((refNum=OpenResFile("\pSampleCode")) != -1)

{

        taskID = DynamicDownload (slotNum, Resource,
        &st_parmblock, startParmSeg,
        lenParmSeg);

        if (taskID != 0)

        {

                /*  You can send and receive messages to this task */



        .

        .

        .

        }

        .

        .

        .

        closeFile (refNum);

}
```

Chapter 9 A/ROSE Prep

A/ROSE Prep is the driver that provides services to Macintosh programs or processes that are used to communicate with other processes on the Macintosh or on one or more smart cards. A/ROSE Prep includes A/ROSE message passing, task naming, and echo services; it is *not* another operating system for the Macintosh computer.

This chapter describes where to find A/ROSE Prep on the MCP distribution disks, how to install and use A/ROSE Prep, and how to make specific calls to A/ROSE Prep. ■

The A/ROSE Prep software

The A/ROSE Prep file contains information that is used by the A/ROSE download subroutines (NewDownLoad and DynamicDownload). Some of this information is specific to different types of cards. The A/ROSE Prep file must be in the system folder before any programs are downloaded onto NuBus cards. Installation instructions are provided in Chapter 2.

There is a unique Board ID for each type of NuBus card. These board Ids are assigned by Apple. The Slot Manager information on the cards is used to determine the Board ID of each card in the system. The Board ID is used as the resource ID to get card specific information; such as routines to start/halt card.

A/ROSE Prep software consists of the A/ROSE Prep driver, development tools, include files, and examples. The MCP distribution disks contains a folder named :A/ROSE Prep: that contains the following:

- a file named `IPCGlue.o` that contains the A/ROSE Prep library, providing object routines (glue code) for interfacing to the A/ROSE Prep driver, as well as glue code that allows C programs running under the Macintosh II operating system to make calls to the driver
- a file named A/ROSE Prep, which contains
  - □ the A/ROSE Prep driver, which runs under the Macintosh II operating system
  - □ an INIT31 resource, which installs the driver and managers at system start-up
  - □ the Name Manager, which is provided for the Macintosh II main logic board
  - □ the Echo Manager, which is provided for the Macintosh II main logic board
- a folder named `Examples`, which contains
  - □ an A/ROSE Prep file that contains everything just described for the A/ROSE Prep file, plus the Echo example. (The Echo example is almost identical to the Echo Manager, and is provided to show how you can add a manager.)
  - □ a makefile that shows how `IPCGlue.o` is used in linking
  - □ Example files that contain source code examples of Macintosh II programs that use the A/ROSE Prep driver

Each of these components is described in this chapter in the section on A/ROSE Prep services, along with examples of C and assembly-language macros for each A/ROSE Prep call.

Using A/ROSE Prep

An application that uses A/ROSE Prep must make an initial call to `OpenQueue` to establish its use of IPC. Each process that uses A/ROSE Prep requests that a queue be opened by calling `OpenQueue`.

Messages are sent and received through A/ROSE Prep using `Send` and `Receive`.

- When the A/ROSE Prep driver gets a `Receive` request and no completion routine is specified, the message queue is searched for a message matching the criteria specified. If a matching message is found, it is returned to the process. If no matching message is found, the driver either returns immediately or, depending on the timeout specified, blocks the process until a matching message arrives (indefinitely if the timeout is 0, or until the timeout is reached).

  However, the `Receive` request behaves differently when a completion routine is specified. Refer to information on the `Receive` call in the next section of this chapter for more details.

- If a `Send` request is destined for a process on the Macintosh II, the destination process is unblocked, if waiting for the message that has arrived, or the message is placed in its queue. If the message is destined for a task on a smart card, the message is transferred to the ICCM on that slot for delivery to the task.

A/ROSE Prep services

This section describes the A/ROSE Prep services and provides examples of how to call primitives and utilities from both C and Assembler. These services are provided to support features similar to those of A/ROSE for applications running on the Macintosh II computer.

◆ *Note:* As with A/ROSE, A/ROSE Prep uses C calling conventions, and all registers are preserved except D0, D1, A0, and A1. Calls in both C and Assembler take arguments and use similar data structures.

Table 9-1 lists the services provided by A/ROSE Prep, with a brief description of each.

■ **Table 9-1** A/ROSE Prep services

Name	Description
`CloseQueue()`	Closes an A/ROSE Prep queue
`FreeMsg()`	Frees a message buffer
`GetCard()`	Returns the NuBus slot number on which the calling process is running
`GetETick()`	returns the number of major ticks
`GetICCTID()`	Returns the task identifier of the InterCard Communication Manager
`GetIPCg()`	Returns the address of the global data area within the A/ROSE Prep driver
`GetMsg()`	Gets a message buffer
`GetNameTID()`	Returns the task identifier of the Name Manager
`GetTickPS()`	Returns the number of major ticks in one second
`GetTID()`	Returns the task identifier of the calling process
`IsLocal()`	Returns an indication of the locality of an address
`KillReceive()`	Cancels an outstanding receive request
`Lookup_Task()`	Returns the task identifier of the process or task that matches the Object Name and Type Name specified
`OpenQueue()`	Opens an A/ROSE Prep queue
`Receive()`	Receives a message
`Register_Task()`	Allows a process to register itself with the Object Name and Type Name specified
`Send()`	Sends a message
`SwapTID()`	Swaps the `mFrom` and `mTo` fields in a message buffer
`LockRealArea()`	Accepts a virtual address and a length; attempts to lock the memory associated with the virtual address into real memory
`UnLockRealArea()`	Unlocks a memory area that was previously locked with a call to LockRealArea()
`NetCopy()`	Takes virtual addresses for its address arguments

CloseQueue( )

`CloseQueue()` closes a queue that was previously opened. This call should be the last one made before an entity terminates.

The C declaration for `CloseQueue()` is

```
        void    CloseQueue();
```

The following example shows how to call `CloseQueue` using assembly language:

```
        JSR     CloseQueue
```

FreeMsg( )

`FreeMsg()` frees a message buffer that was acquired earlier by a call to `GetMsg()`.

The number of messages initially available depends upon the number requested in the named A/ROSE Prep resource entries of type `aipn` found in the A/ROSE Prep driver file.

The C declaration of `FreeMsg()` is

```
void            FreeMsg( mptr )
message         *mptr;                  /* pointer to message buffer to free */
```

The form for the `FreeMsg` macro is as follows, where `P1` is the address of the message buffer to be freed:

```
[Label]         FreeMsg                 P1
```

`P1` can be specified as a register (A0-A6, D0-D7), or use any 68000 addressing mode valid in an LEA instruction to specify the location containing the desired address.

■ Table 9-2 A/ROSE Prep Address Usage

Call	Address Usage
`FreeMsg`	The address of the message buffer to be freed must be a 32-bit virtual address.
`GetIPCg`	The address of the global table will be a 32-bit virtual address.
`GetMsg`	The address of the allocated message buffer will be a 32-bit virtual address.
`IsLocal`	The address to check must be a NuBus address.
`LockRealArea`	The address of the memory to be locked must be a 32-bit virtual address. The address of the addressareas structure must be either a 24-bit virtual or 32-bit virtual address, depending on the current addressing mode of the Macintosh main logic board (see SwapMMUMode).. The addressareas structure will be filled with NuBus address/length pairs.
`Lookup_Task`	The addresses of the type and object name strings must be either 24-bit virtual or 32-bit virtual addresses, depending on the current addressing mode of the Macintosh main logic board (see SwapMMUMode).
`NetCopy`	The source and destination addresses must be 32-bit virtual addresses.
`OpenQueue`	The address of the blocking Receive procedure must be either a 24-bit virtual or 32-bit virtual address, depending on the current addressing mode of the Macintosh main logic board (see SwapMMUMode).
`Receive`	The address of the completion routine must be either a 24-bit virtual or 32-bit virtual address, depending on the current addressing mode of the Macintosh main logic board (see SwapMMUMode). The address of the received message will be a 32-bit virtual address.

Register_Task — The addresses of the type and object name strings must be either 24-bit virtual or 32-bit virtual addresses, depending on the current addressing mode of the Macintosh main logic board (see SwapMMUMode)

Send — The address of the message buffer to send must be a 32-bit virtual address

UnlockRealArea — The address of the memory to unlock must be a 32-bit virtual address

GetCard( )

`GetCard()` returns the NuBus slot number of the card on which the calling process is running. For the Macintosh II computer, the number returned is always zero.

The C declaration for `GetCard()` is

```
        char    GetCard  ();
```

The following example shows how to call `GetCard` using assembly language. Upon return, D0 contains the NuBus slot number on which the calling process is running.

```
        JSR     GetCard
```

GetETick( )

`GetETick()` returns the number of major ticks—that is, the elapsed time in ticks—since the operating system started.

The C declaration for `GetETick()` is

```
        unsigned  long        GetETick();
```

The following example shows the how to call `GetETick` using assembly language. To return the number of major ticks, get the value of location `gMajorTick` in the `gCommon` data area.

```
        JSR     GetETick
```

◆ *Note:* A tick on the Macintosh II is of a different duration than that on an MCP card.

GetICCTID( )

`GetICCTID()` returns the task identifier of the InterCard Communication Manager.

The C declaration for `GetICCTID()` is

```
        tid_type        GetICCTID ();
```

The following example shows the how to call `GetICCTID` using assembly language. Upon return, D0 contains the task identifier of the InterCard Communication Manager.

```
        JSR     GetICCTID
```

◆ *Note:* Slot 0 has an implicit ICCM, since the ICCM is built into the A/ROSE Prep driver that is configured into the System File.

GetIPCg( )

GetIPCg() returns the address of the data area of the A/ROSE Prep driver. This routine is provided as an aid for debugging purposes. Refer to the include files on the MCP distribution disks for the structure of IPCg.

The C declaration for GetIPCg() is

```
        struct IPCg *GetIPCg();
```

▲ **Warning** Use this call at your own risk! Subject to change with no notice. ▲

The following example shows how to call GetIPCg using assembly language. Upon return, D0 contains the address of the data area of the A/ROSE Prep driver.

```
        JSR    GetIPCg
```

◆ *Note:* If you use this routine in Assembler, the routine returns the beginning of the driver's area; you must change the address by an offset defined in IPCgdefs.a in order to use the record for this data area.

GetMsg( )

`GetMsg()` requests a message buffer from the free-message pool. `GetMsg()` either returns zero indicating failure to obtain a message buffer, or a pointer to the allocated message. A call to `FreeMsg()` releases the message.

All fields in the message, except message ID (`mID`) and the From address (`mFrom`), are cleared before the pointer to the message is returned. Message ID is set to a number that is statistically unique to the field; the From address is set to the current task identifier.

The C declaration of `GetMsg()` is

```
        message        *GetMsg();
```

The form for the `GetMsg` macro is

```
        [Label]        GetMsg
```

The address of the allocated message buffer is returned in D0 unless no buffer was available. In that case, 0 is returned in D0.

LockRealArea( )

`LockRealArea()` will accept as input a virtual address and a length and will attempt to lock the memory associated with this virtual address into real memory. The virtual address must be in the same address space as the routine that called `LockRealArea()`. Therefore, a NuBus card cannot lock memory on the mother board. Similarly, another NuBus card and the mother board cannot lock memory on any other NuBus card. If successful, the physical address(es) of the memory associated with the virtual address are returned.

The structure `addressareas`, in which `LockRealArea` returns the physical address/length pair is defined in the following way:

```
struct addressareas

{

      void *address;    /* Physical address of memory area */

            unsigned long length;  /* Length of memory area */

};
```

The calling sequence for `LockRealArea()` is the following:

```
short LockRealArea(void *virtualaddr,unsigned long length,

struct addressareas *buffer,
unsigned long count);
```

virtualaddr is the virtual address of the memory area to be mapped. length is the length of the memory area. buffer is the area where the physical address map is returned. buffer is a pointer to an array of structure addressareas. count is the number of real address/length pairs in the structure addressareas that the buffer can hold. This is the same as the number of elements in the array addressareas.

You can declare the buffer in the following way:

```
struct addressareas buffer[16];
```

If the size of the buffer [] is large enough for only one entry; that is, count has a value of one, the pages are forced to be contiguous. Otherwise, the pages may not to be contiguous when locked in memory.

The physical address/length pairs are returned in buffer. Any unused address/length entries in the buffer are initialized to zero.

LockRealArea() returns zero if successful. If the pages could not be locked in physical memory or if the buffer was not large enough to contain the entire physical address map then LockRealArea() will either return an error code of "erLockFailed" or an error code returned by the Macintosh operating system. The memory is not locked if an error is returned.

GetNameTID( )

`GetNameTID()` returns the task identifier of the Name Manager.

The C declaration for `GetNameTID()` is

```
tid_type     GetNameTID ();
```

The following example shows how to call `GetNameTID` using assembly language. Upon return, D0 is the task identifier of the Name Manager.

```
JSR    GetNameTID
```

GetTickPS( )

`GetTickPS()` returns the number of major ticks in 1 second.

The C declaration for `GetTickPS()` is

```
        unsigned short      GetTickPS ();
```

The following example provides how to call `GetTickPS` using assembly language. Upon return, D0 is the number of major ticks in 1 second.

```
        JSR     GetTickPS
```

GetTID( )

`GetTID()` returns the task identifier of the calling task.

The C declaration for `GetTID()` is

```
        tid_type      GetTID ();
```

The following example shows how to call `GetTID` using assembly language. Upon return, D0 is the task identifier of the calling process.

```
        JSR    GetTID
```

IsLocal( )

`IsLocal()` returns a true or false indication of whether or not an address is local.

The C declaration for `IsLocal()` is

```
short          IsLocal(address)
char           *address;           /* address to test. */
```

`IsLocal()` returns true (non-zero) if the address passed is local. `IsLocal()` returns false ø if the address passed is a remote NuBus address.

The form for the `IsLocal` macro is as follows, where `P1` is the address to examine:

```
[Label]        IsLocal             P1
```

`P1` can be specified as a register (`A0-A6, D0-D7`), an immediate (`#<abs-expr>`), or use any 68000 addressing mode valid in an LEA instruction to specify the location of a long word containing the desired value.

KillReceive( )

`KillReceive()` cancels any outstanding `Receive` request for this process. Messages destined for this process are not discarded.

The C declaration for `KillReceive()` is

```
        void    KillReceive();
```

The following example shows how to call `KillReceive` using assembly language:

```
        JSR     KillReceive
```

KillReceive cannot be called from interrupt level. Only receipt completion routines can be called from interrupt level.

Lookup_Task( )

`Lookup_Task()` returns the task identifier of the process or task that matches the Object Name and Type Name specified, or 0 if no matching process or task is found. The wildcard character "=" is allowed. Initially, the index should be set to 0; on subsequent calls, it should be left unchanged.

◆ *Note:* `Lookup_Task()` modifies the variable index. The variable index allows `Lookup_Task()` to find any additional entries that may match the criteria in subsequent calls.

The C declaration for `Lookup_Task()` is

```
tid_type Lookup_Task (object, type, nm_TID, index)
      char          object [];         /* Object Name */
      char          type [];           /* Type Name */
      tid_type      nm_TID;            /* Name Manager Task Identifier */
      unsigned      short *index;      /* Index */
```

The task identifier of the Name Manager is `nm_TID`, and may be obtained by using `GetNameTID()` for Name Managers on the Macintosh II, or by sending an `ICC_GetCards` message to the ICCM for Name Managers on NuBus cards. `Lookup_Task()` returns the task identifier of the first process or task that matches the criteria.

The following code shows how to look up all processes on the main logic board of the Macintosh II computer:

```
      short index;
      tid_type tid;

      index = 0;
      while ((tid = Lookup_Task ("=", "=", GetNameTID (), &index)) > 0)
            printf ("TID %x Found \n", tid);
```

The following example shows how to call `Lookup_Task` from assembly language:

```
      MOVE.W       #0,INDEX           ; initialize index
      PEA          INDEX              ; address of index
      MOVE.L       TID,D0             ; value of tid on stack
      MOVE.L       D0,-(A7)           ; place on stack
      PEA          TYPE_NAME          ; address of type name
      PEA          OBJECT_NAME        ; address of object name
      JSR          Lookup_Task
      ADDQ.W       #16,A7             ; pop the stack
      TST.W        D0                 ; check if found
      BNE.S        D0,XXX             ; jump if found
```

OpenQueue( )

OpenQueue() assigns an A/ROSE Prep queue and returns the TID of the process that called OpenQueue, or zero if no queue could be assigned. This method allows you to set up your own procedure to determine what to do while waiting on a blocking Receive; if you do not want to use this mechanism, use a parameter of zero. This method also lets you decide whether to cancel the outstanding Receive request or discontinue communication with A/ROSE Prep; that is, it is a way of letting you check for operator termination.

This function must be called before any other call to A/ROSE Prep can be made. You can issue either

- an A/ROSE Prep CloseQueue request, or
- a KillReceive request

If the procedure issues an A/ROSE Prep CloseQueue request and returns to the A/ROSE Prep driver, then the driver returns to the outstanding Receive request with a value of 0. Issuing a KillReceive request returns 0 to the Receive request (no message).

The C declaration for OpenQueue() is

```
tid_type  OpenQueue(procedure)
      void (*procedure) (); /* Procedure to execute while waiting */
                               /* for blocking receive to complete. */
```

◆ *Note:* This parameter is required; use 0 if you do not want to call the procedure.

The form for the OpenQueue macro is as follows, where P1 is the address of the procedure to execute while waiting for a blocking receive to complete:

```
      [Label]        OpenQueue     P1
```

P1 can be specified as a register (A0-A6, D0-D7), an immediate (#<abs-expr>), or use any 68000 addressing mode valid in an LEA instruction to specify the location of a long word containing the desired value.

Receive( )

Receive() returns the highest priority message from the message queue of the process that matches the specified criteria.

The C declaration of Receive() is

```
message             *Receive( mID, mFrom, mCode, timeout, compl )
      unsigned      long  mID;      /* Unique message ID to wait on        */
      tid_type      mFrom;          /* Sender address to wait on     */
      unsigned      short mCode;    /* Message code to wait on             */
      long          timeout;        /* Time to wait in major ticks   */
                                    /* before giving up                    */
      void          compl();        /* Address of a completion routine */
```

The first three parameters (mID, mFrom, and mCode) are selection criteria used to receive a specific kind of message. These parameters may be set to match either a specific value, to match any value (by specifying OS_MATCH_ALL), or to match no value (by specifying OS_MATCH_NONE).

The fourth parameter is the timeout value. A timeout value of 0 waits forever for a satisfying message. A negative value returns either a satisfying message or 0 immediately, and a positive value waits that many ticks for a satisfying message to arrive.

◆ *Note:* If a completion routine is not specified, the A/ROSE Prep Receive performs in exactly the same way as the A/ROSE Receive primitive.

The fifth parameter is the address of a C completion routine. This parameter is required for A/ROSE Prep, and changes the way the Receive request performs. This fifth parameter must be either the address of a completion routine or zero, if no completion routine is desired. When this completion routine parameter is non-zero, the call to Receive always returns immediately with a result of 0 or an error status as shown in Table 9-4.

The completion routine will be called with a parameter of type 'message *'. If the completion routine is passed a pointer of zero, a timeout occurred.

◆ *Note:* It is possible for the completion routine to be called before the Receive actually returns. The purpose of the completion routine is to provide a mechanism by which the Macintosh II application can continue to execute without having to wait for a message. This is necessary because the current version of the Macintosh II operating system is not a multi-tasking operating system; therefore, the application cannot cease to process events. Under A/ROSE, a process can do a blocking Receive and permit other processes to execute.

Table 9-3 describes the results from various settings of the timeout parameter in major ticks for the Receive call. The results column describes what is returned to the Receive request and completion routine, as well as when the completion routine is called.

■ **Table 9-3** State table for the `Receive` call

Timeout value	Completion routine	Message available	Immediate results	Subsequent results
<0	No (0)	No	Returns 0 to the `Receive` request	None
	No (0)	Yes	Returns message to `Receive` request	None
	Yes	No	The A/ROSE Prep driver returns 0 to the `Receive` request; the completion routine is not called	None
	Yes	Yes	The A/ROSE Prep driver calls the completion routine with the message, after which the driver returns 0 to the `Receive` request	None
=0	No (0)	No	Waits until it gets a message, then returns a message to the `Receive` request	Waits for a message; the `OpenQueue` routine is called continuously
	No (0)	Yes	When a message arrives, returns a message to the `Receive` request	None
	Yes	No	Returns 0 to the `Receive` request When a message arrives, the driver calls the completion routine with the message	None
	Yes	Yes	Returns a message to the completion routine and returns 0 to the `Receive` request	None
>0	No (0)	No	Waits for a message that does not arrive If the time interval that you specify expires, then it returns 0 to the `Receive` request	`OpenQueue` routine is called continuously
	No (0)	Yes	Message returns to the `Receive` request	None

continued...

Timeout value	Completion routine	Message available	Immediate results	Subsequent results
	Yes	No	Immediately returns 0 to the Receive request and the task continues executing When a message comes in, the driver calls the completion routine with the message If the timeout expires, the driver calls the completion routine with 0	None
	Yes	Yes	Returns a message to the completion routine; returns 0 to the Receive request	None

When using completion routine, you should observe the following guidelines:

- Never use a blocking Receive in a completion routine.
- Be cautious about starting the next asynchronous Receive within a completion routine, as recursion can be deadly.
- Remember that completion routines might sometimes be called as the result of an interrupt; anticipate the unexpected!

Only one Receive may be outstanding on a given queue at a time; attempted additional Receive routines will return errors. Receive returns 0 in the event of one of the following:

- no message is available (either timeout or non-blocking)
- a negative error code in the case of an error
- or a positive pointer to the received message buffer

◆ *Note:* You must exercise caution when testing the pointer returned by Receive for a negative value to ensure that the test is valid.

The form for the Receive macro is:

```
[Label]       Receive       P1, P2, P3, P4, P5
```

where P1 is the message ID match code, as follows:

```
P2 =  the sender address match code
P3 =  the message code match code
P4 =  the timeout code
P5 =  the completion routine address
```

P1 through P5 can each be specified as a register (A0-A6, D0-D7), an immediate (#<abs-expr>), or any 68000 addressing mode valid in an LEA instruction to specify the location of a long word containing the desired value.

▲ **Warning** Code running on the Macintosh main board must call the A/ROSE Prep driver `Send` or `Receive` *only* when that code is running in a virtual memory (VM) safe mode.

The A/ROSE Prep driver will call all completion routines in VM safe mode as long as the A/ROSE Prep Prep driver `Send` and `receive` are done in VM safe mode.

Completion routines called by the A/ROSE Prep driver as the result of a receive request with completion routine specified can be called either in user or in supervisor mode, in main line code or at interrupt level. Do not assume that completion routines will always be called at interrupt level in supervisor mode.. ▲

Results returned

Whenever you call the `Receive` request on A/ROSE Prep, you get one of three results returned from the IPC driver:

- 0
- message
- negative number (indicating an error)

Table 9-4 lists the two errors only that can be returned when a `Receive` request is made to A/ROSE Prep.

- **Table 9-4** Errors returned

Error	Number	Description
`NoQueueErr`	`-64`	Error code for no more queues or bad queue
`QueueBusy`	`-65`	If `Receive` is already outstanding on queue

Error -64 (`NoQueueErr`) is returned if the queue number (TID) of the task doing the `Receive` request is bad. A queue number is bad if it is not within the range of legal queue numbers or is not open (either `OpenQueue` was not done or `CloseQueue` was done).

Error -65 (`QueueBusy`) is returned if an attempt is made to do a `Receive` request for a particular queue number (TID) when a request is already outstanding. Refer to the section earlier in this chapter on `OpenQueue` for more information.

▲ **Warning** To check for an error in the message pointer returned by a `Receive` request in C language, you *must* cast the message pointer to long before checking to see if the pointer is negative. Failure to do so will result in a system crash. ▲

The following code checks the message pointer to see if an error code was returned:

```
message *msgptr;
msgptr = Receive (0, 0, 0, 0, 0);
if  ((long) msgptr < 0)
{
       /* Process error code */
}
else
{
       /* No error, process message */
}
```

Register_Task( )

`Register_Task()` allows a process to register itself with the Object Name and Type Name specified, using the Name Manager. If the process should be visible only to other processes on the Macintosh II main logic board, `local_only` is set non-zero. If the process should be seen by tasks on other cards, then `local_only` should be set to 0. `Register_Task()` returns a non-zero value if the process was registered; if not, 0 is returned.

The C declaration for `Register_Task()` is

```
typedef boolean short;
        char Register_Task ( object, type, local_only);
        char            object [];          /* Object Name */
        char            type [];            /* Type Name */
        boolean         local_only;         /* If Local Visibility Only */
```

The following code provides an example of how to register a process:

```
if (!Register_Task ("my_name", "my_type", 0))
        printf("Could not Register Process");
```

The following example shows how to call `Register_Task` from assembly language:

```
        MOVE.L          #LOCAL, -(A7)       ; value of local on stack
        PEA             TYPE_NAME           ; address of type name
        PEA             OBJECT_NAME         ; address of object name
        JSR             Register_Task
        ADDQ.W          #12,A7              ; pop the stack
        TST.B           D0                  ; check if register ok
        BNE.S           OK                  ; jump if OK
```

Send( )

Send() allows you to send a message to the destination address specified in the message. Send() places a message on the queue of the process specified by the message field, mTo. The message is placed in the queue in priority order (from highest to lowest). It is assumed that all fields have been filled in (mFrom, mTo, mCode, and so forth) when this call is made.

The C declaration of Send() is

```
void            Send( mptr )
message         *mptr;                  /* pointer to message buffer */
```

If a message is undeliverable, it is returned to the sender with the message status, mStatus, set to 0x8000 and the message code, mCode, having bit 1 << 15 set.

The assembly-language form for the Send macro is as follows, where P1 is the address of the message buffer to be sent:

```
[Label]         Send                    P1
```

P1 can be specified as a register (A0-A6, D0-D7), or can use any 68000 addressing mode valid in an LEA instruction to specify the location containing the address of the message buffer to be sent.

SwapTID( )

SwapTID() swaps the mFrom and mTo fields of a message buffer.

The C declaration of SwapTID() is

```
        void SwapTID( mptr )
        message       *mptr;              /* pointer to message buffer */
```

The assembly-language form for the SwapTID macro is as follows, where P1 is the address of the message buffer:

```
        [Label]        SwapTID        P1
```

P1 can be specified as a register (A0-A6, D0-D7), or can use any 68000 addressing mode valid in an LEA instruction to specify the location containing the desired address.

UnlockRealArea( )

UnlockRealArea() unlocks a memory area that was previously locked with a call to LockRealArea().

The calling sequence for UnlockRealArea() is the following:

```
short UnlockRealArea(void *virtualaddress, unsigned long length);
```

virtualaddress is the beginning virtual address of an area of memory that was previously locked. length is the length of the memory area that was locked.

UnlockRealArea() returns zero if the address was successfully unlocked. Otherwise, an error is returned.

◆ *Note:* The address and length parameters specified in a call to UnlockRealArea() must exactly match an address and length specified in a call to LockRealArea(). UnlockRealArea() cannot handle fragmented unlocking in this release; in other words, you cannot unlock a portion of a perviously locked memory area.

UnlockRealArea() returns a status of zero if the virtual memory support is not available.

NetCopy( )

NetCopy() exists in both A/ROSE running on a smart NuBus card and the A/ROSE Prep driver running on the Macintosh main board. NetCopy() is a solution to many problems involving virtual memory.

NetCopy() takes virtual addresses for its address arguments. The NetCopy() call will then examine internal A/ROSE structures to determine if it can convert the virtual addresses to *real*

NuBus addresses. These internal structures are initizlized when A/ROSE on the smart NuBus card and the A/ROSE Prep driver on the Macintosh main board initialize. The internal structures are then updated if LockRealArea or UnlockRealArea is done on the Macintosh main board in a virtual memory environment.

If NetCopy cannot convert a *virtual* address to a *real* NuBus address, NetCopy sends an internal A/ROSE message to a task (located in the appropriate virtual address space) that can perform the conversion. A message is returned to NetCopy when the conversion is completed.

LockRealArea is important in the function of NetCopy. When converting virtual addresses to *real* NuBus addresses, NetCopy examines the internal structures updated by LockRealArea and UnlockRealArea . NetCopy does not have to send internal A/ROSE messages if the buffers were locked down using LockRealArea.

◆ *Note:* NetCopy will never be as fast as using MapNuBus and doing the copying directly. However, NetCopy is much safer than using MapNuBus.

`NetCopy()` will copy data from a source virtual address to a destination virtual address. `NetCopy()` has been designed to be fail safe and users are advised to use only `NetCopy` for all their data transfers. If the memory areas specified are locked in memory then the copy process will go very fast.

▲ **Warning** NetCopy may send messages and issue blocking receive requests to wait for replies. Therefore, `NetCopy` must not be called at interrupt level or code that must be run in run-to-block mode, or by code on the tick or idle chain. ▲

The calling sequence for `NetCopy()` is the following:

```
short NetCopy(tid_type srcTID, void *srcAddress,

              tid_type dstTID, void *dstAddress,

              unsigned long bytecount);
```

The address `srcAddress` is a virtual address as viewed by the task whose Task ID is `srcTID`. The address `dstAddress` is a virtual address as viewed by the task whose Task ID is `dstTID`.

`NetCopy()` will safely, perhaps slowly, copy data from the source to the destination. Both the source and destination addresses can be paged out to disk in a virtual memory environment. `NetCopy()` will cause these pages to be brought back into the physical memory and perform the copy. The copying of data can be done by the main board rather than by the NuBus card in some cases.

`NetCopy()` returns zero if the copy was successful. Otherwise, `NetCopy()` returns an error status.

Error Codes:

Non zero if there was an error in `NetCopy()`.

▲ **Warning** `srcAddress` and `dstAddress` must both be 32 bit clean addresses. Memory manager flags must not be in the high byte of a mother board address.

Do not call `NetCopy` from interrupt routine because it does a blocking receive. Do not call `NetCopy` in idle chain because you cannot block idle chain. ▲

If you call `NetCopy` from task that runs under run to block mode, be aware that `NetCopy` may do a receive and give up the control of the CPU.

Chapter 10 **Using the Forwarder with A/ROSE Prep**

The Forwarder is a general purpose dangling entity (an unassociated piece of code) for providing A/ROSE tasks the ability to access AppleTalk. This chapter describes the Forwarder, tells how the Forwarder sends messages in conjunction with A/ROSE Prep, provides instructions on installing the Forwarder, lists the messages and errors codes used by the Forwarder, and provides example code. ■

What is the Forwarder?

The Forwarder is a mechanism for the interchange of messages between tasks running on MCP-based cards under A/ROSE and applications over the AppleTalk network system; the Forwarder communicates via the AppleTalk Data Stream Protocol (ADSP). (For more information on ADSP and other AppleTalk protocols, refer to *Inside AppleTalk*.) Both multiple server tasks and requests from multiple client applications can be handled by the Forwarder.

The Forwarder functions as a gateway, converting ADSP messages to A/ROSE messages. *Figure 10-1* shows the message path when a client machine sends data over the AppleTalk network system to the server. A **server** is a NuBus-compatible Macintosh computer with an MCP-based smart card installed. A **client machine** is any Macintosh computer that incorporates code in its application to use the Forwarder. Both the server and client are part of the AppleTalk network system.

The data travels over the AppleTalk network system though the main logic board on the Macintosh II to communicate with the task running on the MCP card.

How the Forwarder sends messages

The Forwarder sends messages when:

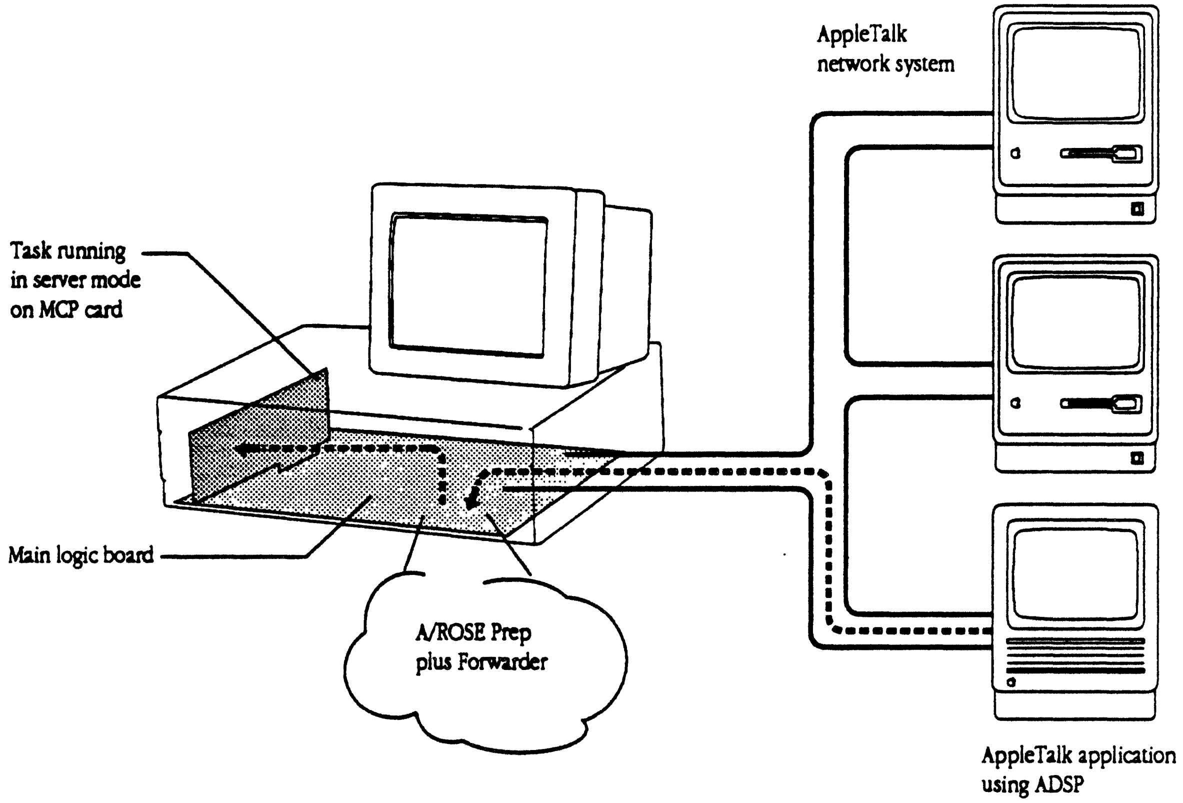
- a task running under A/ROSE on an MCP card wants to send data to an application on another machine over the AppleTalk network system
- an application running on a machine on the AppleTalk network system wants to send data to a task running under A/ROSE

The following figures show the processing sequence using the Forwarder when an application running on a client machine wants to send a message to an MCP card (the server) over the AppleTalk network system.

Within the file `FWD` are two resources that can be used for configuring the Forwarder:

- `svcn`, which tells the Forwarder how much memory to preallocate for the server and for communications. The Forwarder will attempt to call for this number of free services and free (validate?) communication memory available.
- `sysz`, which can be changed to increase the size of the system heap. For more information, refer to the section about the INIT Resource 31 in *Inside Macintosh*, Volume 5, "System Startup Information".

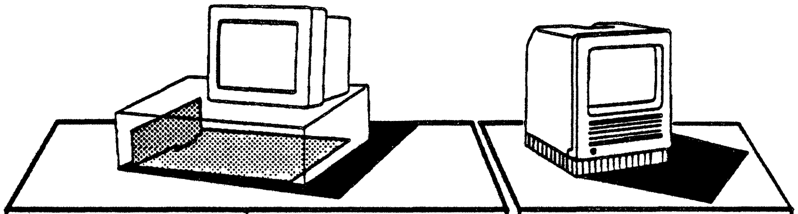
■ **Figure 10-1** Message paths using the Forwarder


AppleTalk network system
Task running in server mode on MCP card
Main logic board
A/ROSE Prep plus Forwarder
AppleTalk application using ADSP

Initialization

Figure 10-2 lists the initialization process for the Forwarder, the server, and the client respectively.

■ **Figure 10-2** Initialization process using the Forwarder



Initialization process using the Forwarder	**Server** A/ROSE on MCP card	**Forwarder** on the Macintosh II	**Client** on the AppleTalk network system
		Mac II boots up; Forwarder registers name with A/ROSE system Name Manager	
End of Forwarder initialization			
	MCP card gets loaded; server starts executing on card Uses Name Manager `Lookup_Task()` to request to find Forwarder Sends `MC_OPENSERVER` to Forwarder to register name on AppleTalk *Wait*		
End of Server initialization			
	Server open for business		Issues an NBP `Look_Up` for servers on the AppleTalk network system Opens an ADSP connection to the Forwarder via ADSP driver request
End of Client initialization			

normal processing follows...

The Forwarder registers its name with the Name Manager using the `Register_Task()` routine using the Object Name `"Forwarder"` and Type Name `"ADSP"`. The server task issues an A/ROSE Name Manager `Lookup_Task()` request to find the TID of the Forwarder .

The server task then registers its name with the Forwarder with an `MC_OPENSERVER` call, which the Forwarder acknowledges. The Forwarder then registers the server's name using the Name Binding Protocol (NBP) `Look_Up` call (refer to *Inside AppleTalk* for more information). The application on a client machine finds the Forwarder also using the NPB `Look_Up` call.

Normal processing with the Forwarder

Figure 10-3 illustrates normal processing using the Forwarder. This set of messages are repeated as long as the server and client want to communicate with each other.

The application on a client machine on the network initiates a connection to the Forwarder using ADSP; the application then sends a message (or messages) to the Forwarder. The Forwarder generates a Connection ID to identify the ADSP connection when the connection is established.

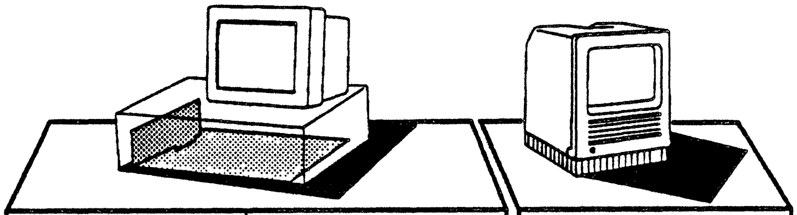
The Forwarder then sends the message to the server using the `MC_READDATA` message code and waits for a reply from the server. At this point, the server knows the Connection ID (which identifies the client application).

◆ *Note:* Messages are sent one at a time in either direction. Before a second message can be sent, the sender must wait for an acknowledgement. There can be one `MC_READDATA` and one `MC_SENDDATA` outstanding per connection at any one time.

The server prepares a reply and sends it back to the Forwarder in an `MC_SENDDATA` message code, after which the Forwarder sends `MC_SENDDATA+1` to reply to the server. The Forwarder then sends the message over the AppleTalk network system to the requesting application on the client machine.

◆ *Note:* The server can send data acknowledgement (`MC_READDATA+1`) either before or after the server sends data using `MC_SENDDATA`, depending on how code for the server and client is written.

■ **Figure 10-3** Normal processing using the Forwarder



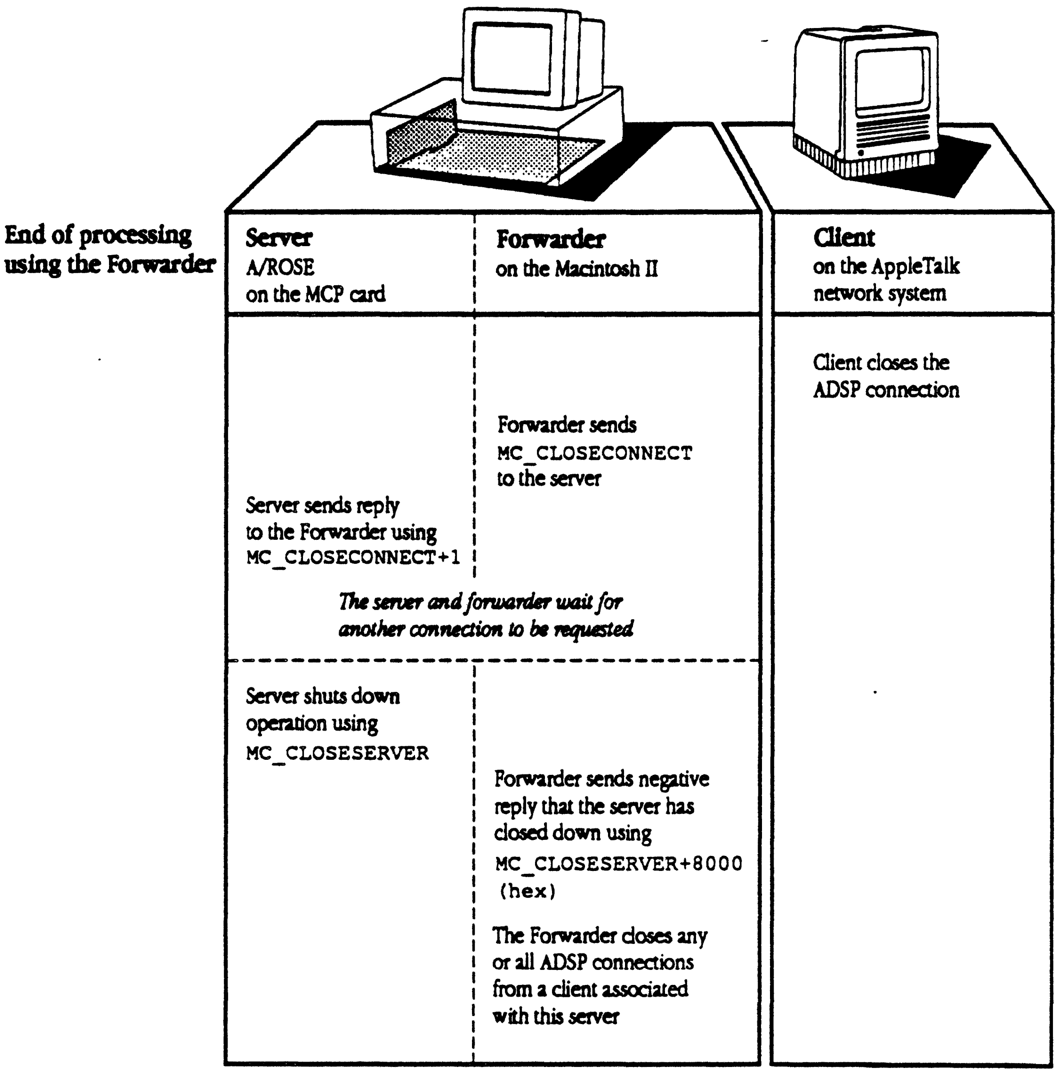
Normal processing using the Forwarder	**Server** A/ROSE on MCP card	**Forwarder** on the Macintosh II	**Client** on the AppleTalk network system
			Client sends data via ADSP to the Forwarder
		Sends data to server using `MC_READDATA`	
	Server sends data to client via the Forwarder using `MC_SENDDATA`		
	Server sends acknowledgement of receiving data from the Forwarder `MC_READDATA+1`		
		Forwarder sends the data to the client via ADSP	Client receives data
		Forwarder sends reply to server using `MC_SENDDATA+1`	

end of processing follows...

Completing communication with the Forwarder

Figure 10-4 shows how the client completes communication and terminates the connection.

- **Figure 10-4** End of processing using the Forwarder



End of processing using the Forwarder

Server A/ROSE on the MCP card

Forwarder on the Macintosh II

Client on the AppleTalk network system

Client closes the ADSP connection

Forwarder sends MC_CLOSECONNECT to the server

Server sends reply to the Forwarder using MC_CLOSECONNECT+1

The server and forwarder wait for another connection to be requested

Server shuts down operation using MC_CLOSESERVER

Forwarder sends negative reply that the server has closed down using MC_CLOSESERVER+8000 (hex)

The Forwarder closes any or all ADSP connections from a client associated with this server

The client closes the ADSP connection; the Forwarder sends MC_CLOSECONNECT to the server; the Server sends a reply to the Forwarder using MC_CLOSECONNECT+1. The server and Forwarder wait for another connection to be requested.

At any point, the server can discontinue its availabillity by sending an MC_CLOSESERVER message to the Forwarder. The server acknowledges that the server has closed down, and closes any or all ADSP connections from a client that are associated with this server. In actuality, the server and Forwarder wait continuously for more connections from other clients.

Using the Forwarder

This section describes how to install the Forwarder, lists the messages you need to use the Forwarder, and provides examples of code to use the Forwarder for both the client machine and server machine in the transaction.

This section also describes the errors returned by the Forwarder.

Installing the Forwarder

The Forwarder is code that resides in memory on the Macintosh II above BufPtr. This code is provided on the MCP distribution disk in the file :Forwarder:FWD. The Forwarder is installed by an INIT31 resource (in the same manner as the A/ROSE Prep driver) during start-up of the Macintosh.

To install the Forwarder:

1. Move the file FWD into the System Folder of the Macintosh II.
2. The Forwarder uses both the A/ROSE Prep driver and ADSP, an Init file. Place both of these files in the System Folder of your Macintosh II at this time, if you have not already done so.
3. Reboot.

Messages used by the Forwarder

Table 10-1 lists the messages used by the Forwarder, and the direction in which the message is sent. Each of these messages is more fully described after the table.

- **Table 10-1** Messages used by the Forwarder

Name	Direction of message
MC_CLOSECONNECT	Forwarder to the server
MC_CLOSESERVER	Server to the Forwarder
MC_ECHO	Forwarder to the server
MC_OPENSERVER	Server to the Forwarder
MC_READDATA	Forwarder to the server
MC_SENDDATA	Server to the Forwarder

MC_CLOSECONNECT

The Forwarder uses the MC_CLOSECONNECT message to tell the server task that the specified client has closed the connection with the server.

The message parameters for MC_CLOSECONNECT are as follows:

```
mCode          MC_CLOSECONNECT
mOData[0]      Connection Identifier
mOData[1]      Reason connection closed
                   = 1 (if connection failed)
                   = 0 (if connnection closed normally)
```

The reply parameter for MC_CLOSECONNECT is as follows:

```
mCode          MC_CLOSECONNECT + 1
```

MC_CLOSESERVER

The MC_CLOSESERVER message is sent by the server task to the Forwarder to tell the Forwarder that the server is shutting down. The Forwarder closes all connections and unregisters the server's name on the Appletalk network system.

The message parameter for MC_CLOSESERVER is as follows:

```
mCode          MC_CLOSESERVER
```

The reply parameter for MC_CLOSESERVER is as follows:

```
mCode          MC_CLOSESERVER + 1
```

MC_ECHO

The MC_ECHO message is sent from the Forwarder to all server tasks every 30 seconds to test if the servers are still running. Each server must reply to the message to let the Forwarder know it is active.

The message parameter for MC_ECHO is as follows:

```
mCode          MC_ECHO
```

The reply parameter for MC_ECHO is as follows:

```
mCode          MC_ECHO + 1
```

MC_OPENSERVER

The `MC_OPENSERVER` message is sent from a server to the Forwarder to tell the Forwarder to register its name on the Appletalk network system begin accepting ADSP connections on the server's behalf.

The message parameters for `MC_OPENSERVER` are as follows:

`mCode`	`MC_OPENSERVER`
`mFrom`	Server's task ID (used by Forwarder to uniquely identify the servers)
`mDataPtr`	Pointer to NBP EntityName structure with type, object, and zone filled in
`mDataSize`	size of EntityName structure

The reply parameters for `MC_OPENSERVER` are as follows:

`mCode`	`MC_OPENSERVER + 1`
`mStatus`	Result

MC_READDATA

The `MC_READDATA` message is sent by the Forwarder to the server task when data is received from a client. The data is not copied onto the server's card. The server must use `CopyNuBus` to access the data. The server *must* reply to this message to free up the Forwarder's data space and receive further messages.

◆ *Note:* The connection ID is first given to the server using `MC_READDATA`. When the server gets a connection ID that it does not recognize, the server knows it is a new connection and should do any connection initialization necessary (for example, adding the ID to a list of open connections).

The message parameters for `MC_READDATA` are as follows:

`mCode`	`MC_READDATA`
`mFrom`	Forwarder's task ID
`mDataPtr`	Pointer to data
`mDataSize`	Length of data
`mOData[0]`	Connection ID
`mOData[1]`	End-of-message flag =1 (if end of message) =0 (not end of message)

◆ *Note:* Refer to ADSP documentation for more information on the end-of-message flag.

The reply parameters for `MC_READDATA` are as follows:

`mCode`	`MC_READDATA + 1`
`MOData[0]`	`Connection ID`

MC_SENDDATA

The MC_SENDDATA message is sent by the server to send data to ADSP clients. The Forwarder will send a reply to this message when it is able to accept more data from the server.

The message parameters for MC_SENDDATA are as follows:

mCode	MC_SENDDATA
mFrom	Server's task ID
mDataPtr	Pointer to data
mDataSize	Length of data (limited to DATA_BUFFER, which is 580 bytes)
mOData[0]	Connection ID
mOData[1]	End-of-message flag =1 (if end of message) =0 (not end of message)

◆ *Note:* Refer to ADSP documentation for more information on the end-of-message flag.

The reply parameters for MC_SENDDATA are as follows:

mCode	MC_SENDDATA + 1
mOData[0]	Connection ID
mStatus	Result

Using the Forwarder on the server machine

Following is an example of a task that gets downloaded to the MCP card; this example shows how the server establishes a server task and uses the Forwarder. The task must send the MC_OPENSERVER message to the Forwarder; the server uses the Type Name and Object Name to register its name on the AppleTalk network system.

◆ *Note:* On the lines highlighted in bold, you should use your own Type Name and Object Name.

```
/*
 *     FWDExample.c -      A/ROSE forwarder example.
 *
 *     Copyright © 1988, 1989 Apple Computer, Inc. All rights reserved.
 *
 *     In this example, the server receives data from a client. The
 *     server changes uppercase letters to lowercase letters, and lowercase
 *     letters to uppercase letters, then sends the data back to the client
 *     using the Forwarder.
 */

#include "os.h"
#include "managers.h"
```

```
#include  "arose.h"
#include  "siop.h"
#include  "AppleTalk.h"
#include  "ADSP.h"
#include  "FWD.h"

pascal  void  illegal()
       extern        0x4afc;

tid_type      fwd_tid;

FWDExample()
{
       message       *msg;
       long          finish;
       short         done;
       EntityName    ent;
       long          mid;



       printf("FWD Example starting.\n");

       fwd_tid = GetFWDTID();           /*    Get the forwarder task ID  */

       if  (!fwd_tid)
       {
              printf("Couldn't find forwarder.\n");
              StopTask( GetTID() );
       }

       /*     Fill in NBP entity structure      */

       ent.objStr.length = 4;
       BlockMove(  "TYPE",  ent.objStr.text,  5  );
                            /*  Enter  your  Type  Name       */
       ent.typeStr.length = 7;
       BlockMove(  "OBJECT",  ent.typeStr.text,  8  );
                            /*  Enter  your  Object  Name     */

       ent.zoneStr.length = 1;
       BlockMove( "*", ent.zoneStr.text, 2 );

       /*     Send OPENSERVER request to forwarder    */

       msg = GetMsg();
       msg->mTo = fwd_tid;
       msg->mCode = MC_OPENSERVER;
       msg->mDataPtr = &ent;
       msg->mDataSize = sizeof(EntityName);
       mid = msg->mId;
       Send( msg );
```

```
msg = Receive( 0, 0, MC_OPENSERVER+1, 0 );
if (msg->mStatus)
{
        printf("MC_OPENSERVER failed.status = %x\n",
                msg->mStatus);
        FreeMsg( msg );
        StopTask( GetTID() );
}
FreeMsg( msg );

finish = GetETick() + 5 * 60 * GetTickPS();
                                /*      Stick Around for 5 minutes             */
done = 0;
while(!done)
{
        msg = Receive( 0, 0, 0, finish - GetETick() );
                                /* Wait for message or timeout */
        if (msg)
                switch(msg->mCode)
                {
                        case    MC_CLOSECONNECT:
                                /* Connection Failed/Closed */
                                printit( "CLOSECONNECT", msg );
                                Reply( msg, 0 );
                                break;
                        case    MC_READDATA:    /* Received data from
                                                /* client via FWD*/
                                printit( "READDATA", msg );
                                dosomething( msg );
                                Reply( msg, 0 );        /* Let forwarder know */
                                                /* we've read the data     */
                                break;
                        case    MC_SENDDATA+1:/* write data(from the */
                                /* server to the client) */
                                printit( "SENDDATA (Reply)", msg );
                                FreeMem( msg->mDataPtr ); /* Forwarder */
                                        /* is done with buffers; free them */
                                FreeMsg( msg );
                                break;
                        case    MC_ECHO:        /* Tickle Message */
                                printit( "ECHO", msg );
                                Reply( msg, 0 );        /* Let forwarder know */
                                        /* we are alive */
                                break;
                        default:
                                printit( "BAD", msg );
                                Reply( msg, 0x8000 );
                                break;
                }
        else    /*      if timeout      */
        {
                msg = GetMsg();
```

```
                        msg->mTo = fwd_tid;
                        msg->mCode = MC_CLOSESERVER;
                        Send( msg );
                        done = 1;
                }
        }
        printf("FWD example finished!\n");
}
        /* This example was adapted from GetPrintTID in printf.c */

static tid_type GetFWDTID ()
{
        tid_type                FWDTID;
        struct ra_GetCards  get_cards;
        message                 *msgptr;
        short                   index;
        short                   s;

        FWDTID = 0;

        if (GetICCTID () != 0)
        {
                if ((msgptr = GetMsg ()) == NULL)
                        return (FWDTID);

                msgptr -> mCode = ICC_GETCARDS;
                msgptr -> mDataPtr = &get_cards;
                msgptr -> mDataSize = sizeof (struct ra_GetCards);
                msgptr -> mTo = GetICCTID ();
                Send (msgptr);

                msgptr = Receive (OS_MATCH_ALL, OS_MATCH_ALL, ICC_GETCARDS+1,
                                OS_NO_TIMEOUT);

                if (msgptr -> mStatus == 0)
                {
                        for (s = 0; (s < IC_MAXCARDS) && (FWDTID == 0); s++)
                        {
                                if (get_cards.tid[s] > 0)
                                {
                                        index = 0;
                )               FWDTID = Lookup_Task ("Forwarder", "ADSP",
                                        get_cards.tid[s], &index);
                                        printf("FWDTID=%x; NMTID=%x\n", FWDTID,
                                                get_cards.tid[s] );
                                }
                        }
                }
                FreeMsg (msgptr);
        }
        else
        {
                index = 0;
                FWDTID = Lookup_Task ("Forwarder", "ADSP", GetNameTID (),
                                &index);
                printf("Local: FWDTID=%x; NMTID=%x\n", FWDTID, GetNameTID() );
        }

        return (FWDTID);
}

printit( what, msg )
char            *what;
message         *msg;
{
        printf("---- %s\n", what );
        printf(
        "mId = %08.8X        mCode = %04.4X       mStatus = %04.4X
        mPriority = %04.4X\n", msg->mId, msg->mCode, msg->mStatus,
        msg->mPriority );
        printf("    mFrom = %08.8X    mTo = %08.8X    mDataPtr = %08.8X
        mDataSize = %08.8X\n", msg->mFrom, msg->mTo, msg->mDataPtr, msg-
>mDataSize );
        printf("    mSData[0] = %08.8X     mSData[1] = %08.8X
                mSData[2] = %08.8X\n", msg->mSData[0], msg->mSData[1],
                msg->mSData[2] );
        printf("    mOData[0] = %08.8X     mOData[1] = %08.8X
                mOData[2] = %08.8X\n", msg->mOData[0], msg->mOData[1],
                msg->mOData[2] );
}

        /* Replace the following with your code to process data */

dosomething(msg)
message         *msg;
{
        char            buffer[100];
        message         *m;
        short   i;


        CopyNuBus( msg->mDataPtr, buffer, msg->mDataSize+1 );

        printf("---- Data Received: %s\n", buffer );

        for( i=0; i < msg->mDataSize; i++ )
                if ((buffer[i] >= 'a') && (buffer[i] <= 'z'))
                        buffer[i] = buffer[i] - 'a' + 'A';
                else if ((buffer[i] >= 'A') && (buffer[i] <= 'Z'))
                        buffer[i] = buffer[i] - 'A' + 'a';

        printf("---- Data Sent: %s\n", buffer );

        /*      Send processed data to client      */
```

```
      m = GetMsg();
      m->mTo = fwd_tid;
      m->mDataPtr = GetMem( msg->mDataSize );
      m->mDataSize = msg->mDataSize;
      BlockMove( buffer, m->mDataPtr, msg->mDataSize+1 );
      m->mCode = MC_SENDDATA;
      m->mOData[0] = msg->mOData[0];
      m->mOData[1] = 1;         /*    EOM flag     */
      Send( m );
}

void
Reply( m, stat )
message     *m;
unsigned short          stat;
{
      tid_type    temp;

      if (m)
      {
            if (m->mStatus == 0x8000)
                  FreeMsg( m );
            else
            {
                  temp = m->mFrom;
                  m->mFrom = m->mTo;
                  m->mTo = temp;
                  m->mStatus = stat;
                  if (m->mStatus == 0x8000)
                        m->mCode |= 0x8000;
                  else
                        m->mCode |= 1;
#ifdef      DEBUG
                  printf("Sending reply (%x) to %x, status = %x\n",
                        m->mCode, m->mTo, m->mStatus );
#endif      DEBUG
                  Send( m );
            }
      }
}
```

Using the Forwarder from the client machine

The following is an example of MPW code that you can put in your client application. The first line shows the command to run the code; the second two lines show the output on a client machine on the network.

```
FWDExample TYPE OBJECT "ThIS is ThE FiRSt EXampLe"

Sending 'ThIS is ThE FiRSt EXampLe'
Received 'tHis IS tHe fIrsT exAMPlE'
```

The following code shows the source code for the MPW tool to access the server on an MCP card. (This code is currently not on the MCP distribution disks.)

```
/*
 *     FWDExample.c -      MPW Tool to access "server" on MCP card.
 *
 *     Copyright © 1988, Apple Computer, Inc. All rights reserved.
 *
 *     The tool is accessed by:
 *
 *     FWDExample    TYPE OBJECT "MESSAGE"
 *
 *     TYPE is the NBP type that the server has registered as.
 *     OBJECT is the NBP object that the server has registered as.
 *     MESSAGE is the data that the server is to act upon.
 *
 */

#include     "Types.h"
#include     "stdio.h"
#include     "Memory.h"
#include     "ADSP.h"
#include     "AppleTalk.h"

#define      USAGE         "# FWDExample type object \"message\"\n"
#define      Q_SIZE        200                 /* Size of our ADSP queues
       */
#define      WEIRD_SIZE    200          /* NBP wants big buffer for some reason
*/

short               dspRefNum;          /* ADSP ref. num. from OpenDriver
       */
EntityName          ent;                /* NBP entity name  */
char                adr[WEIRD_SIZE];    /* AddrBlock buffer */
short               count;                     /* Number of nodes found by NBP */
TPCCB               ccb;                /* ADSP Connection Control Block */
Ptr                 sendQ, recvQ, attn; /* ADSP queues      */
DSPPBPtr            openPB;             /* Open parameter block */
DSPParamBlock       pb;                        /* Param block for ADSP requests */
short               rc;                 /* Place to put result codes      */
char                buffer[200];        /* Buffer for processed data */
```

```
main(argc,argv)
int             argc;
char    *argv[];
{
        short  MyLookupName();

        if (argc < 3)
        {
                fprintf( stderr, "## Not enough parameters.\n");
                fprintf( stderr, USAGE );
                exit(1);
        }
        if ((strlen(argv[1]) < 1) || (strlen(argv[1]) > 30)) /* TYPE */
        {
                fprintf( stderr, "## \"type\" must be from 1 to 30 characters
                in length.\n");
                fprintf( stderr, USAGE );
                exit(1);
        }
        if ((strlen(argv[2]) < 1) || (strlen(argv[2]) > 30)) /* OBJECT */
        {
                fprintf( stderr, "## \"object\" must be from 1 to 30 characters
                in length.\n");
                fprintf( stderr, USAGE );
                exit(1);
        }
        if ((strlen(argv[3]) < 1) || (strlen(argv[3]) > 100))
                /* MESSAGE    */
        {
                fprintf( stderr, "## \"message\" must be from 1 to 100
                characters in length.\n");
                fprintf( stderr, USAGE );
                exit(1);
        }

        /* open MPP first */

        if ((rc = MPPOpen()) != noErr)
        {
                fprintf( stderr, "MPP Open failed. err=%d\n", rc);
                fprintf( stderr, USAGE );
                exit(1);
        }

        /* open ADSP */

        if ((rc = OpenDriver(".DSP", &dspRefNum)) != noErr)
        {
                fprintf( stderr, "ADSP Open failed. err=%d\n",rc);
                fprintf( stderr, USAGE );
                exit(1);
        }
```

```
/* allocate ADSP pointers */

sendQ = NewPtr(Q_SIZE);
if (sendQ == 0L)
{
        fprintf( stderr, "Memory failed.\n");
        fprintf( stderr, USAGE );
        exit(1);
}

recvQ = NewPtr(Q_SIZE);
if (recvQ == 0L)
{
        fprintf( stderr, "Memory failed.\n");
        fprintf( stderr, USAGE );
        exit(1);
}

attn = NewPtr(attnBufSize);
if (attn == 0L)
{
        fprintf( stderr, "Memory failed.\n");
        fprintf( stderr, USAGE );
        exit(1);
}

ccb = (TPCCB)NewPtr(sizeof(TRCCB));
if (ccb == 0L)
{
        fprintf( stderr, "Memory failed.\n");
        fprintf( stderr, USAGE );
        exit(1);
}

openPB = (DSPPBPtr)NewPtr(sizeof(DSPParamBlock));
if (openPB == 0L)
{
        fprintf( stderr, "Memory failed.\n");
        fprintf( stderr, USAGE );
        exit(1);
}

/*      Fill in Entity block to be passed to NBPLookup */

ent.objStr.length = strlen (argv[1]);
BlockMove( argv[1], ent.objStr.text, ent.objStr.length+1 );
ent.typeStr.length = strlen(argv[2]);
BlockMove( argv[2], ent.typeStr.text, ent.typeStr.length+1 );
ent.zoneStr.length = 1;
BlockMove( "*", ent.zoneStr.text, ent.zoneStr.length+1);
count = 0;
```

```
if (!MyLookupName( &ent, adr, 10, 5, &count ))
        /* Find our server */
{
        fprintf( stderr, "## Lookup failed.\n");
        fprintf( stderr, USAGE );
        exit(1);
}

if (count < 1)        /* If none found */
{
        fprintf( stderr, "## Couldn't find the server.\n");
        fprintf( stderr, USAGE );
        exit(1);
}

/* Initialize connection end */

pb.ioCompletion = 0L;
pb.ioVRefNum = 0;
pb.ioCRefNum = dspRefNum;
pb.csCode = dspInit;
pb.u.initParams.ccbPtr = ccb;
pb.u.initParams.userRoutine = 0L;
pb.u.initParams.sendQSize = Q_SIZE;
pb.u.initParams.sendQueue = sendQ;
pb.u.initParams.recvQSize = Q_SIZE;
pb.u.initParams.recvQueue = recvQ;
pb.u.initParams.attnPtr = attn;
pb.u.initParams.localSocket = 0;
rc = PBControl(&pb, false);
if (rc != noErr)
{
        fprintf( stderr, "## ADSP Init failed.\n err=%d", rc);
        fprintf( stderr, USAGE );
        exit(1);
}

/*      Request a connection */

openPB->ioCompletion = 0L;
openPB->ioVRefNum = 0;
openPB->ioCRefNum = dspRefNum;
openPB->csCode = dspOpen;
openPB->ccbRefNum = ccb->refNum;
(openPB->u.openParams.remoteAddress).aNet = ((AddrBlock *)adr)->aNet;
(openPB->u.openParams.remoteAddress).aNode = ((AddrBlock *)adr)->aNode;
(openPB->u.openParams.remoteAddress).aSocket = ((AddrBlock *)adr)->aSocket;
(openPB->u.openParams.filterAddress).aNet = 0;
(openPB->u.openParams.filterAddress).aNode = 0x00;
(openPB->u.openParams.filterAddress).aSocket = 0x00;
openPB->u.openParams.ocMode = ocRequest;
```

```
openPB->u.openParams.ocInterval = 4;
openPB->u.openParams.ocMaximum = 4;
rc = PBControl(openPB, false);
if (rc != noErr)
{
       fprintf( stderr, "## ADSP Open failed. err=%d\n", rc);
       fprintf( stderr, USAGE );
       exit(1);
}

fprintf( stderr, "Sending '%s'\n", argv[3] );


/* Send data to server */

pb.ioCompletion = 0L;
pb.ioVRefNum = 0;
pb.ioCRefNum = dspRefNum;
pb.csCode = dspWrite;
pb.u.ioParams.reqCount = strlen(argv[3]) + 1;
pb.u.ioParams.dataPtr = argv[3];
pb.u.ioParams.eom = 1;
pb.u.ioParams.flush = 1;                 /* flush now */
rc = PBControl(&pb, false);
if (rc != noErr)
{
       fprintf( stderr, "## ADSP Write failed. err=%d\n", rc);
       fprintf( stderr, USAGE );
       exit(1);
}

/* Read processed data from server */

pb.ioCompletion = 0L;
pb.ioVRefNum = 0;
pb.ioCRefNum = dspRefNum;
pb.csCode = dspRead;
pb.ccbRefNum = ccb->refNum;
pb.u.ioParams.reqCount = 101;
pb.u.ioParams.dataPtr = buffer;
rc = PBControl(&pb, false);
if (rc != noErr)
{
       fprintf( stderr, "## ADSP Read failed. err=%d\n", rc);
       fprintf( stderr, USAGE );
       exit(1);
}

fprintf( stderr, "Received '%s'\n", buffer );

/* Close ADSP connection */
```

```
        pb.ioCompletion = 0L;
        pb.ioVRefNum = 0;
        pb.ioCRefNum = dspRefNum;
        pb.csCode = dspRemove;
        pb.ccbRefNum = ccb->refNum;
        rc = PBControl(&pb, false);
        if (rc != noErr)
        {
                fprintf( stderr, "## ADSP Remove failed. err = %d\n", rc);
                fprintf( stderr, USAGE );
                exit(1);                /* arbitrary exit code */
        }

        /* deallocate ADSP pointers */

        DisposPtr(sendQ);
        DisposPtr(recvQ);
        DisposPtr(attn);
        DisposPtr(openPB);
        DisposPtr(ccb);

        /* close ADSP driver */

        if (CloseDriver(dspRefNum) != noErr)
        {
                fprintf( stderr, "## ADSP Close failed. err = %d\n", rc);
                fprintf( stderr, USAGE );
                exit(1);
        }

        exit(0);
}

short
MyLookupName(srvrEnt, adrBufPtr, interval, count, numgotten)
EntityName          *srvrEnt;
char                *adrBufPtr;
short               interval, count;
short               *numgotten;
{
        NBPparms    nbp;
        char        entBufPtr[200];
        OSErr       rc;

        /* set up entity */
        NBPSetEntity(entBufPtr, &(srvrEnt->objStr),
                &(srvrEnt->typeStr), &(srvrEnt->zoneStr));

        /* look for specified server */
        nbp.interval = interval;
        nbp.count = count;
        nbp.parm.Lookup.retBuffPtr = adrBufPtr;
        nbp.parm.Lookup.retBuffSize = WEIRD_SIZE;
        nbp.parm.Lookup.maxToGet = 1;
        nbp.NBPPtrs.entityPtr = entBufPtr;

        rc = PLookupName(&nbp, false);
        if (rc != noErr)
        {
                fprintf( stderr, "Lookup failed. err=%d\n", rc);
                fprintf( stderr, USAGE );
                return(false);
        }

        /* return number found */

        *numgotten = nbp.parm.Lookup.numGotten;

        return(true);

}
```

Message transactions while the Forwarder is active

The following shows the flow of A/ROSE messages between a typcial server and the Forwarder before, during, and after the transaction.

```
FWD Example starting.

FWDTID=4; NMTID=2
----     ECHO
         mId = 00005BF7      mCode = 2002    mStatus = 0000      mPriority = 0000
         mFrom = 00000004    mTo = 0B000003  mDataPtr = 00000000
         mDataSize = 00000000    mSData[0] = 00000000    mSData[1] = 00000000
         mSData[2] = 00000000    mOData[0] = 00000000    mOData[1] = 00000000
         mOData[2] = 00000000
----     ECHO
         mId = 00005BFB      mCode = 2002    mStatus = 0000      mPriority = 0000
         mFrom = 00000004    mTo = 0B000003  mDataPtr = 00000000
         mDataSize = 00000000    mSData[0] = 00000000    mSData[1] = 00000000
         mSData[2] = 00000000    mOData[0] = 00000000    mOData[1] = 00000000
         mOData[2] = 00000000
----     READDATA
         mId = 00005BFC      mCode = 1006    mStatus = 0000      mPriority = 0000
         mFrom = 00000004    mTo = 0B000003  mDataPtr = 00026AA0
         mDataSize = 0000001A    mSData[0] = 00000000    mSData[1] = 00000000
         mSData[2] = 00000000    mOData[0] = 00000002    mOData[1] = 00000001
         mOData[2] = 00000000
----     Data Received: ThIS is ThE FiRSt EXampLe
----     Data Sent: tHis IS tHe fIrsT exAMPlE
----     SENDDATA (Reply)
         mId = FB00003A      mCode = 1009    mStatus = 0000      mPriority = 0000
         mFrom = 00000004    mTo = 0B000003  mDataPtr = FB06DF18
         mDataSize = 0000001A    mSData[0] = 00000000    mSData[1] = 00000000
         mSData[2] = 00000000    mOData[0] = 00000002    mOData[1] = 00000001
         mOData[2] = 00000000
----     CLOSECONNECT
         mId = 00005BFD      mCode = 1004    mStatus = 0000      mPriority = 0000
         mFrom = 00000004    mTo = 0B000003  mDataPtr = 00000000
         mDataSize = 00000000    mSData[0] = 00000000    mSData[1] = 00000000
         mSData[2] = 00000000    mOData[0] = 00000002    mOData[1] = 00000000
         mOData[2] = 00000000
----     ECHO
         mId = 00005C01       mCode = 2002   mStatus = 0000      mPriority = 0000
         mFrom = 00000004    mTo = 0B000003  mDataPtr = 00000000
         mDataSize = 00000000    mSData[0] = 00000000    mSData[1] = 00000000
         mSData[2] = 00000000    mOData[0] = 00000000    mOData[1] = 00000000
         mOData[2] = 00000000
----     ECHO
         mId = 00005C05      mCode = 2002    mStatus = 0000      mPriority = 0000
         mFrom = 00000004    mTo = 0B000003  mDataPtr = 00000000
         mDataSize = 00000000    mSData[0] = 00000000    mSData[1] = 00000000
         mSData[2] = 00000000    mOData[0] = 00000000    mOData[1] = 00000000
         mOData[2] = 00000000
```

```
----   ECHO
       mId = 00005C09      mCode = 2002    mStatus = 0000      mPriority = 0000
       mFrom = 00000004    mTo = 0B000003  mDataPtr = 00000000
       mDataSize = 00000000    mSData[0] = 00000000    mSData[1] = 00000000
       mSData[2] = 00000000    mOData[0] = 00000000    mOData[1] = 00000000
       mOData[2] = 00000000
----   ECHO
       mId = 00005C0D      mCode = 2002    mStatus = 0000      mPriority = 0000
       mFrom = 00000004    mTo = 0B000003  mDataPtr = 00000000
       mDataSize = 00000000    mSData[0] = 00000000    mSData[1] = 00000000
       mSData[2] = 00000000    mOData[0] = 00000000    mOData[1] = 00000000
       mOData[2] = 00000000
---- ECHO
       mId = 00005C11      mCode = 2002    mStatus = 0000      mPriority = 0000
       mFrom = 00000004    mTo = 0B000003  mDataPtr = 00000000
       mDataSize = 00000000    mSData[0] = 00000000    mSData[1] = 00000000
       mSData[2] = 00000000    mOData[0] = 00000000    mOData[1] = 00000000
       mOData[2] = 00000000
----   ECHO
       mId = 00005C15      mCode = 2002    mStatus = 0000      mPriority = 0000
       mFrom = 00000004    mTo = 0B000003  mDataPtr = 00000000
       mDataSize = 00000000    mSData[0] = 00000000    mSData[1] = 00000000
       mSData[2] = 00000000    mOData[0] = 00000000    mOData[1] = 00000000
       mOData[2] = 00000000
----   ECHO
       mId = 00005C19      mCode = 2002     mStatus = 0000     mPriority = 0000
       mFrom = 00000004    mTo = 0B000003  mDataPtr = 00000000
       mDataSize = 00000000    mSData[0] = 00000000    mSData[1] = 00000000
       mSData[2] = 00000000    mOData[0] = 00000000    mOData[1] = 00000000
       mOData[2] = 00000000
----   ECHO
       mId = 00005C1D      mCode = 2002    mStatus = 0000      mPriority = 0000
       mFrom = 00000004    mTo = 0B000003  mDataPtr = 00000000
       mDataSize = 00000000    mSData[0] = 00000000    mSData[1] = 00000000
       mSData[2] = 00000000    mOData[0] = 00000000    mOData[1] = 00000000
       mOData[2] = 00000000
FWD example finished!
```

Errors returned by the Forwarder

Table 10-2 lists the errors returned by the Forwarder, and briefly describes each.

■ **Table 10-2** Errors returned by the Forwarder

Error	Description
FWE_DupServer	Only one server can be opened per A/ROSE task. (Attempted to open more than one.)
FWE_NoServer	Forwarder did not find a server registered under the current task ID.
FWE_Write	Attempted to issue an MC_SENDDATA before the Forwarder was finished processing the previously issued MC_SENDDATA for this connection.
FWE_NoConnect	The Connection ID specified was not found.
FWE_Overflow	The maximum data size of 580 bytes (DATA_BUFFER) was exceeded by MC_SENDDATA.
FWE_NoSMemory	Could not get memory on the Macintosh II to open server.
FWE_NoSListen	The ADSP listener failed.
FWE_NoRegister	The NBP name registration failed. (This error most likely occurred due to a duplicate server name, or improperly-filled in EntityName structure.)

Chapter 11 **Troubleshooting Guide**

THIS CHAPTER describes the illegal instructions and debugger calls that can occur when using A/ROSE and the A/ROSE Prep driver, and lists error codes and messages that may be returned for both A/ROSE and the A/ROSE Prep driver.

This chapter assumes you have a working knowledge of the M68000 microprocessor architecture and instruction set. ■

What happened?

During development, you computer system will crash or hang from time to time. Here's what to do when either of those situations occur:

- If the system crashes,look at the load map of the code executing on the card, or at the supervisor stack for the Macintosh II.
- If the system hangs, you must "hunt and discover" to find where there is a possible problem in the code. On a smart card, check the task control blocks; on the Macintosh II, check the supervisor stack.

◆ *Note:* To find the task control blocks, check the pointer named `gTaskTable` in the array within `gCommon`.

The sections that follow may help you determine what has occurred and provide direction for correcting the problem. A/ROSE troubleshooting is discussed first, followed by A/ROSE Prep troubleshooting.

Troubleshooting A/ROSE

If the operating system code on the MCP smart card appears to have stopped running, A/ROSE may have crashed or may be in a hung state.

Where do you start troubleshooting? The value `gCommon.gMajorTick` provides an indication of whether or not the A/ROSE kernel is still functioning. The value `gCommon.gMajorTick` is the major tick counter within A/ROSE, and is incremented at the beginning of every major tick cycle.

If `gCommon.gMajorTick` is incrementing, the system has not crashed, but may be hung. Go to the section in "A/ROSE hangs" called "gCommon.gMajorTick Is Incrementing".

If `gCommon.gMajorTick` is not incrementing, A/ROSE may either have hung, detected a problem and intentionally crashed by executing an illegal instruction, or crashed due to an exception (such as a bus error). The following information will help determine if A/ROSE has crashed.

On execution of an exception or hardware error interrupt, an A/ROSE handler dumps the current register set to the "crash area," a portion of card memory starting at 0x0600 on the smart card. *Table 11-1* lists the format of the crash area.

■ **Table 11-1** Crash area format

Memory Location	+0	+4	+8	+C
0x0600	D0	D1	D2	D3
0x0610	D4	D5	D6	D7
0x0620	A0	A1	A2	A3
0x0630	A4	A5	A6	SSP
0x0640	SR PC	USP	Flag	
0x0650	trap number			

where

`SSP` is the Supervisor Stack Pointer

`SR` is the Status Register

`PC` is the Program Counter

`USP` is the User Stack Pointer

`Flag` is a byte that starts at address 0x064A that contains the value 0xFF when an error has occurred. Clearing this byte causes the registers to be reloaded with the saved registers and the system restarted.

`trap number` is the 68000 exception ID

Examine the `Flag` byte at 0x064A. If it contains an 0xFF, the system has crashed; go to the section on If A/ROSE Crashes. Otherwise, the system is hung; go to the section on A/ROSE Hangs to determine the cause of the hang.

◆ *Note:* When `Flag` is 0, this area of memory has *no* meaning. Specifically, this area of memory does not show the current registers or state of anything when this `Flag` is 0.

Using dumpcard

To assist in troubleshooting during your development efforts, you can use the MPW tool dumpcard to display a list of values within the A/ROSE.

Dumpcard dumps the card and formats the output to the standard output you specify in MPW.

The syntax of the dumpcard tool is the following:

```
dumpcard [-∂?] [-b] [-d fwa lwa] [-e] [-f] [-h] [-m] [-n] [-r] [-s s1 s2 ... sn] [-
t] [-v].
```

where	`-∂?`	displays description of options
	`-b`	dumps A/ROSE memory blocks
	`-d fwa lwa`	dumps card memory from "fwa" to "lwa" (lwa/fwa both in hex)
	`-e`	do not dump exception vectors
	`-f`	display a list of valid slots
	`-h`	halts a running card
	`-m`	do not dump A/ROSE messages
	`-n`	do not dump names for TIDs
	`-r`	do not dump registers
	`-s s1 s2...sn`	dump cards in specified slots (Default is no slots dumped)
	`-t`	do not dump task info
	`-v`	show dumpcard version

The following example shows how to dump the contents of an MCP smart card in slot B. Use the MPW tool dumpcard in the folder :A/ROSE:Examples:MCP and enter the following string in the MPW worksheet:

```
dumpcard -s b -e
```

In this example, the following information would be sent to the standard output you specify in MPW (such as the Macintosh II screen).

```
Version 1.1.1(RWM/GAB).

•••••  Slot #E



Object Name      Type Name        Task ID

name manager     name manager     e000201

time manager     time manager     e000302

RSM              RSM              e000504

echo manager     echo manager     e000605

echo example     echo manager     e000807

Trace Manager    Trace Manager    e000706


Unable to display registers - processor running


Exception Vectors

00000000: Reset (Initial SSP)           FE080000

00000004: Reset (Initial PC)            FE00961C

00000008: Bus Error                     FE004CD4

0000000C: Address Error                 FE004CE0

00000010: Illegal Instruction           FE004CEC

00000014: Zero Divide                   FE004CF8

00000018: CHK Instruction               FE004D04

0000001C: TRAPV Instruction             FE004D10

00000020: Privilege Violation           FE004D1C
```

```
00000024: Trace                              FE004D28
00000028: Line 1010 Emulator                 FE004D34
0000002C: Line 1111 Emulator                 FE004D40
0000003C: Uninitialized Interupt Vector      FE004D70
00000060: Spurious Interrupt                 FE004DDC
00000064: Level 1 Interrupt                  FE00091E
00000068: Level 2 Interrupt                  FE00893A
0000006C: Level 3 Interrupt                  FE004E00
00000070: Level 4 Interrupt                  FE004E0C
00000074: Level 5 Interrupt                  FE004E18
00000078: Level 6 Interrupt                  FE004E24
0000007C: Level 7 Interrupt                  FE004E30
00000080: Trap 0  (Unused5)                  FE004AB2
00000084: Trap 1  (Unused4)                  FE004AB2
00000088: Trap 2  (FreeMsg)                  FE005036
0000008C: Trap 3  (GetMsg)                   FE005118
00000090: Trap 4  (Spl)                      FE0053C6
00000094: Trap 5  (Send)                     FE005374
00000098: Trap 6  (Receive)                  FE005254
0000009C: Trap 7  (StartTask)                FE0053E0
000000A0: Trap 8  (StopTask)                 FE0055A4
000000A4: Trap 9  (Reschedule)               FE005330
000000A8: Trap 10 (SpecReqs)                 FE005174
000000AC: Trap 11 (FreeMem)                  FE004FA8
000000B0: Trap 12 (GetMem)                   FE00506A
000000B4: Trap 13 (Unused2)                  FE004AB2
000000B8: Trap 14 (Unused1)                  FE004AB2
000000BC: Trap 15 (Unused3)                  FE004AB2

*****   Slot #B

Unable to display registers - processor running
```

```
Initial A5 value           FE001D3A

Memory buffer list ptr     FE009718

Slot address               FE000000

Slot number                0E000000

Time Base                  A16D45B4

Major Tick                 00023BAC

CAP (Magic Number)         1CCA1943

CAP (Pointer)              FE00178C

CAP (Checksum)             1ACA30CF

Free message list          FE077892

Unique Counter             FE10CE30

Tick Chain                 FE0085AC

Idle Chain                 FE0085B2

Current Task Pointer       FE07EA98

Idle Loop Counter          0186FBCF

Task Table Pointer         FE07EAF8

Error status               00000000

Timeout queue              FE0715E0

Priority Table Pointer     FE07EF00

Priority List              FE07EFF8

Name Unregister pointer    FE001DAC

FWA of message area        FE0776F8

LWA of message area        FE07E840

Initial PC                 FE00961C

FWA of initial code        FE001DD6

LWA of initial code        FE009714

Minor Tick Counter         00040000

Debugger Pointer           00000000

Debugger Comm Area         00000000

Release version            0101
```

```
Current Task ID             0E000100

Minor / Major Cycle         0008

Ticks per Second            0013

Major Cycles defered        FA49

Major Cycles skipped        0000

Page Latch                  0000

Name Task TID               0E000201

ICC Task TID                0E000403

Trace Task TID              00000000

Timer Task TID              0E000302

Messages discarded          00

Major Flag                  00

Time Queue Flag             00

Debugger Flag               00



Task Table dump


** Task #0 (TCB = FE07EA98)

    Next Task (priority)    FE07EA98

    Next Task (timeout)     00000000

    Stack Buffer            FE07E848

    Heap Buffer             00000000

    Program Counter         FE0044C8

    Stack Pointer           FE07EA50

    Code Segment            00000000

    Data Segment            00000000

    Start Parameters        00000000

    Parent TID              00000000

    Status Register         0004

    Page Latch              0000
```

```
  Priority                  00

  Block Mode                01

  Receive Mode              00


  Task ID                   0000

  Message Q Head            00000000

  Message Q Tail            00000000

  Blocked Timeout Value     00000000

  Blocked Message ID        00000000

  Blocked Message From      00000000

  Blocked Message Code      00000000
 —Stack (TOS 100 bytes)
FE07EA50: 00 00 00 01 00 00 00 01 00 00 00 00 00 00 00 00 ................
FE07EA60: 00 00 00 00 00 02 3B B0 00 00 FF FF 00 00 00 00 ......;.........
FE07EA70: FE 07 EA 98 FE 07 15 E0 FE 07 78 1E FE 00 18 A8 ..........x.....
FE07EA80: FE 00 17 8C FE 00 44 3A 00 00 00 D6 FE 00 4C CA .......:......L.
FE07EA90: FE 07 EA F0 01 00 00 0C FE 07 EA 98 00 00 00 00 ................
FE07EAA0: FE 07 E8 48 00 00 00 00 FE 00 44 C8 FE 07 EA 50 ...H......D....P
FE07EAB0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
FE07EAC0: 00 04 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
FE07EAD0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 01 ................
FE07EAE0: FE 00 44 C2 FE 00 09 2A 0E 00 01 00 00 00 00 00 ..D....*........
FE07EAF0: FE 07 EE F8 01 00 00 81 FE 07 EA 98 FE 07 76 98 ..............v.
FE07EB00: FE 07 65 E8 FE 07 55 38 FE 07 54 08 FE 07 43 58 ..e...U8..T...CX
FE07EB10: FE 07 41 A8 FE 07 3C F8 FE 07 3B 48 FE 07 26 90 ..A...<...;H..&.
FE07EB20: FE 07 15 E0 00 00 00 00 00 00 00 00 00 00 00 00 ................
FE07EB30: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
FE07EB40: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................


** Task #1 (TCB = FE077698)
```

```
Next Task (priority)...FE075538

Next Task (timeout)....00000000

Stack Buffer...........FE076648

Heap Buffer............00000000

Program Counter........FE007180

Stack Pointer..........FE0774F4

Code Segment...........00000000

Data Segment...........00000000

Start Parameters.......00000000

Parent TID.............00000000

Status Register........0004

Page Latch.............0001

Priority...............1F

Block Mode.............01

Receive Mode...........FF


Task ID................0001

Message Q Head.........00000000

Message Q Tail.........FE0776C8

Blocked Timeout Value..00023C4F

Blocked Message ID.....00000000

Blocked Message From...00000000

Blocked Message Code...00000000


D0 00000000  D1 00023C4F  D2 00000000  D3 00000010

D4 000000BE  D5 00023C4F  D6 0000009F  D7 00000010

A0 FE073B48  A1 FE077698  A2 00000000  A3 FE077892

A4 FE0015E6  A5 FE001D3A  A6 FE077688

Running in " o".


—Stack (TOS 100 bytes)
```

```
FE0774F4: 00 00 00 00 00 02 3C 4F 00 00 00 00 00 00 00 10 ......<O........
FE077504: 00 00 00 BE 00 02 3C 4F 00 00 00 9F 00 00 00 10 ......<O........
FE077514: FE 07 3B 48 FE 07 76 98 00 00 00 00 FE 07 78 92 ..;H..v.......x.
FE077524: FE 00 15 E6 FE 00 1D 3A FE 07 76 88 FE 00 59 E0 ........:..v...Y.
FE077534: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 9F ................
FE077544: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
FE077554: 00 00 00 00 00 00 00 00 00 00 00 00 00 01 FE 07 ................
FE077564: 78 92 06 74 65 73 74 65 72 00 00 00 00 00 00 00 x..tester.......
FE077574: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
FE077584: 00 00 00 00 03 70 6D 72 00 00 00 00 00 00 00 00 .....pmr........
FE077594: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
FE0775A4: 00 00 00 00 00 00 01 00 0D 70 72 69 6E 74 20 6D .........print m
FE0775B4: 61 6E 61 67 65 72 00 00 00 00 00 00 00 00 00 00 anager..........
FE0775C4: 00 00 00 00 00 00 00 00 00 00 0D 70 72 69 6E 74 ...........print
FE0775D4: 20 6D 61 6E 61 67 65 72 00 00 00 00 00 00 00 00  manager........
FE0775E4: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 6A ...............j


** Task #2 (TCB = FE0765E8)
  Next Task (priority)...FE075408
  Next Task (timeout)....00000000
  Stack Buffer...........FE075598
  Heap Buffer............00000000
  Program Counter........FE007180
  Stack Pointer..........FE076564
  Code Segment...........00000000
  Data Segment...........00000000
  Start Parameters.......00000000
  Parent TID.............00000000
  Status Register........0004
  Page Latch.............0000
  Priority...............1E
  Block Mode............01
  Receive Mode...........FF

  Task ID................0002
  Message Q Head........00000000
  Message Q Tail.........FE076618
  Blocked Timeout Value..00000000
  Blocked Message ID.....00000000
  Blocked Message From...00000000
  Blocked Message Code...00000000

  D0 00000000  D1 00000000  D2 00000000  D3 00000003
  D4 00000000  D5 00000001  D6 0E000302  D7 00000003
  A0 FE076618  A1 FE0765E8  A2 00000000  A3 00000000
  A4 FE0778CC  A5 FE001D3A  A6 FE0765D8
  Running in " o".

  ---Stack (TOS 100 bytes)
FE076564: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 03 ................
FE076574: 00 00 00 00 00 00 00 01 0E 00 03 02 00 00 00 03 ................
FE076584: FE 07 66 18 FE 07 65 E8 00 00 00 00 00 00 00 00 ..f...e.........
FE076594: FE 07 78 CC FE 00 1D 3A FE 07 65 D8 FE 00 7E 8E ..x....:..e...~.
FE0765A4: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
FE0765B4: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
FE0765C4: 00 00 00 00 00 00 00 00 00 00 00 00 FE 07 77 36 ..............w6
FE0765D4: FE 00 16 C6 00 00 00 00 FE 00 4C CA FE 07 66 40 ..........L...f@
FE0765E4: 01 00 00 0C FE 07 54 08 00 00 00 00 FE 07 55 98 ......T.......U.
FE0765F4: 00 00 00 00 FE 00 71 80 FE 07 65 64 00 00 00 00 ......q...ed....
FE076604: 00 00 00 00 00 00 00 00 00 00 00 00 00 04 00 00 ................
FE076614: 1E FF 00 02 00 00 00 00 FE 07 66 18 00 00 00 00 ..........f.....
```

```
FE076624: 00 00 00 00 00 00 00 00 00 00 00 01 FE 00 1D BC ................
FE076634: FE 00 09 2A 0E 00 03 02 00 00 00 00 FE 07 76 90 ...*..........v.
FE076644: 01 00 02 0A 4F 56 46 4C 00 00 00 00 00 00 00 00 ....OVFL........
FE076654: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................


** Task #3 (TCB = FE075538)
  Next Task (priority)...FE077698
  Next Task (timeout)....00000000
  Stack Buffer...........FE075468
  Heap Buffer............00000000
  Program Counter........FE008154
  Stack Pointer..........FE0754E0
  Code Segment...........00000000
  Data Segment...........00000000
  Start Parameters.......00000000
  Parent TID.............00000000
  Status Register........0004
  Page Latch.............FFFF
  Priority...............1F
  Block Mode.............01
  Receive Mode...........FF

  Task ID................0003
  Message Q Head.........00000000
  Message Q Tail.........FE075568
  Blocked Timeout Value..00000000
  Blocked Message ID.....00000000
  Blocked Message From...00000000
  Blocked Message Code...00000000
```

```
 D0 00000000  D1 00000000  D2 00000000  D3 00000000
 D4 0000000E  D5 0002FFFF  D6 00000001  D7 FA020001
 A0 FE075568  A1 FE075538  A2 FE072D26  A3 FE0017A4
 A4 FE00178C  A5 FE001D3A  A6 00000000
   Running in "B¥ "@2<6 ".

 ---Stack (TOS 100 bytes)
FE0754E0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
FE0754F0: 00 00 00 0E 00 02 FF FF 00 00 00 01 FA 02 00 01 ................
FE075500: FE 07 55 68 FE 07 55 38 FE 07 2D 26 FE 00 17 A4 ..Uh..U8..-&....
FE075510: FE 00 17 8C FE 00 1D 3A 00 00 00 00 00 00 00 00 .......:........
FE075520: 00 00 00 00 00 00 00 00 00 00 00 00 FE 00 4C CA ..............L.
FE075530: FE 07 55 90 01 00 00 0C FE 07 76 98 00 00 00 00 ..U.......v.....
FE075540: FE 07 54 68 00 00 00 00 FE 00 81 54 FE 07 54 E0 ..Th.......T..T.
FE075550: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
FE075560: 00 04 FF FF 1F FF 00 03 00 00 00 00 FE 07 55 68 ..............Uh
FE075570: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 01 ................
FE075580: FE 00 1D C4 FE 00 09 2A 0E 00 04 03 00 00 00 00 .......*........
FE075590: FE 07 65 E0 01 00 02 0A 4F 56 46 4C 00 00 00 00 ..e.....OVFL....
FE0755A0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
FE0755B0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
FE0755C0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
FE0755D0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................


** Task #4 (TCB = FE075408)
  Next Task (priority)...FE074358
  Next Task (timeout)....00000000
  Stack Buffer...........FE0743B8
  Heap Buffer............00000000
  Program Counter........FE007180
```

```
Stack Pointer..........FE07533C
Code Segment...........00000000
Data Segment...........00000000
Start Parameters.......00000000
Parent TID.............00000000
Status Register........0014
Page Latch.............0000
Priority...............1E
Block Mode.............01
Receive Mode...........FF

Task ID................0004
Message Q Head.........00000000
Message Q Tail.........FE075438
Blocked Timeout Value..00000000
Blocked Message ID.....00000000
Blocked Message From...00000000
Blocked Message Code...00000000

D0 00000000  D1 00000000  D2 00000000  D3 00000000
D4 00000000  D5 00000000  D6 00000000  D7 0E000504
A0 FE075438  A1 FE075408  A2 00000000  A3 00000000
A4 00000000  A5 FE001D3A  A6 FE0753F8
Running in " o".

 ---Stack (TOS 100 bytes)
FE07533C: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
FE07534C: 00 00 00 00 00 00 00 00 00 00 00 00 0E 00 05 04 ................
FE07535C: FE 07 54 38 FE 07 54 08 00 00 00 00 00 00 00 00 ..T8..T.........
FE07536C: 00 00 00 00 FE 00 1D 3A FE 07 53 F8 FE 00 6E D8 .......:..S...n.
FE07537C: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
```

```
FE07538C: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
FE07539C: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
FE0753AC: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
FE0753BC: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
FE0753CC: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
FE0753DC: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
FE0753EC: 00 00 00 00 00 00 00 00 FE 00 1D 3A 00 00 00 00 ...........:....
FE0753FC: FE 00 4C CA FE 07 54 60 01 00 00 0C FE 07 43 58 ..L...T`......CX
FE07540C: 00 00 00 00 FE 07 43 B8 00 00 00 00 FE 00 71 80 ......C.......q.
FE07541C: FE 07 53 3C 00 00 00 00 00 00 00 00 00 00 00 00 ..S<............
FE07542C: 00 00 00 00 00 14 00 00 1E FF 00 04 00 00 00 00 ................


** Task #5 (TCB = FE074358)
  Next Task (priority)...FE0741A8
  Next Task (timeout)....00000000
  Stack Buffer...........FE074208
  Heap Buffer............00000000
  Program Counter........FE007180
  Stack Pointer..........FE0742F0
  Code Segment...........00000000
  Data Segment...........00000000
  Start Parameters.......00000000
  Parent TID.............00000000
  Status Register........0004
  Page Latch.............0000
  Priority...............1E
  Block Mode.............00
  Receive Mode...........FF

  Task ID................0005
  Message Q Head.........00000000
  Message Q Tail.........FE074388
  Blocked Timeout Value..00000000
  Blocked Message ID.....00000000
  Blocked Message From...00000000
  Blocked Message Code...00000000

  D0 00000000  D1 00000000  D2 00000000  D3 00000000
  D4 00000000  D5 00000000  D6 00000000  D7 0E000B0A
  A0 FE074388  A1 FE074358  A2 00000000  A3 FE0778CC
  A4 FE0015C4  A5 FE001D3A  A6 00000000
  Running in " o".

  —Stack (TOS 100 bytes)
FE0742F0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
FE074300: 00 00 00 00 00 00 00 00 00 00 00 00 0E 00 0B 0A ................
FE074310: FE 07 43 88 FE 07 43 58 00 00 00 00 FE 07 78 CC ..C...CX......x.
FE074320: FE 00 15 C4 FE 00 1D 3A 00 00 00 00 FE 00 58 D8 .......:......X.
FE074330: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
FE074340: 00 00 00 00 00 00 00 00 00 00 00 00 FE 00 4C CA ..............L.
FE074350: FE 07 43 B0 01 00 00 0C FE 07 41 A8 00 00 00 00 ..C.......A.....
FE074360: FE 07 42 08 00 00 00 00 FE 00 71 80 FE 07 42 F0 ..B.......q...B.
FE074370: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
FE074380: 00 04 00 00 1E FF 00 05 00 00 00 00 FE 07 43 88 ..............C.
FE074390: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
FE0743A0: FE 00 1D 9C FE 00 09 2A 0E 00 06 05 00 00 00 00 .......*........
FE0743B0: FE 07 54 00 01 00 02 0A 4F 56 46 4C 00 00 00 00 ..T.....OVFL....
FE0743C0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
FE0743D0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
FE0743E0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
```

```
** Task #6 (TCB = FE0741A8)
 Next Task (priority)...FE073CF8
 Next Task (timeout)....FE073B48
 Stack Buffer...........FE073D58
 Heap Buffer............00000000
 Program Counter........FE007180
 Stack Pointer..........FE074140
 Code Segment...........00000000
 Data Segment...........00000000
 Start Parameters.......00000000
 Parent TID.............00000000
 Status Register........0008
 Page Latch.............0000
 Priority...............1E
 Block Mode.............00
 Receive Mode...........01

 Task ID................0006
 Message Q Head.........00000000
 Message Q Tail.........FE0741D8
 Blocked Timeout Value..00023BCA
 Blocked Message ID.....FFFFFFFF
 Blocked Message From...FFFFFFFF
 Blocked Message Code...0000FFFF

 D0 FE077698  D1 00023BCA  D2 00000000  D3 00000000
 D4 00000000  D5 00000000  D6 00000000  D7 00000000
 A0 FE0715E0  A1 FE0741A8  A2 00000000  A3 00000000
 A4 FE0014DE  A5 FE001D3A  A6 FE074198
 Running in " o".
 —Stack (TOS 100 bytes)
FE074140: FE 07 76 98 00 02 3B CA 00 00 00 00 00 00 00 00 ..v...;.........
FE074150: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
FE074160: FE 07 15 E0 FE 07 41 A8 00 00 00 00 00 00 00 00 ......A.........
FE074170: FE 00 14 DE FE 00 1D 3A FE 07 41 98 FE 00 3F 8C .......:..A...?.
FE074180: FF FF FF FF FF FF FF FF FF FF FF FF 00 00 00 13 ................
FE074190: 00 00 00 00 00 00 00 00 00 00 00 00 FE 00 4C CA ..............L.
FE0741A0: FE 07 42 00 01 00 00 0C FE 07 3C F8 FE 07 3B 48 ..B.......<...;H
FE0741B0: FE 07 3D 58 00 00 00 00 FE 00 71 80 FE 07 41 40 ..=X......q...A@
FE0741C0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
FE0741D0: 00 08 00 00 1E 01 00 06 00 00 00 00 FE 07 41 D8 ..............A.
FE0741E0: 00 02 3B CA FF FF FF FF FF FF FF FF FF FF 01 00 ...;............
FE0741F0: FE 00 1D 94 FE 00 09 2A 0E 00 07 06 00 00 00 00 .......*........
FE074200: FE 07 43 50 01 00 00 2A 4F 56 46 4C 00 00 00 00 ..CP...*OVFL....
FE074210: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
FE074220: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
FE074230: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................


** Task #7 (TCB = FE073CF8)
 Next Task (priority)...FE0765E8
 Next Task (timeout)....00000000
 Stack Buffer...........FE073BA8
 Heap Buffer............00000000
 Program Counter........FE007180
 Stack Pointer..........FE073C8C
 Code Segment...........00000000
 Data Segment...........00000000
 Start Parameters.......00000000
 Parent TID.............00000000
 Status Register........0014

 Page Latch............0000

 Priority...............1E

 Block Mode............00

 Receive Mode...........FF


 Task ID................0007

 Message Q Head........00000000

 Message Q Tail.........FE073D28

 Blocked Timeout Value..00000000

 Blocked Message ID.....00000000

 Blocked Message From...00000000

 Blocked Message Code...00000000


 D0 00000000  D1 00000000  D2 00000000  D3 00000000

 D4 00000000  D5 00000000  D6 00000000  D7 00000000

 A0 FE073D28  A1 FE073CF8  A2 00000000  A3 00000000

 A4 FE001484  A5 FE001D3A  A6 FE073CE8

 Running in " o".


 —Stack (TOS 100 bytes)

FE073C8C: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................

FE073C9C: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................

FE073CAC: FE 07 3D 28 FE 07 3C F8 00 00 00 00 00 00 00 00 ..=(..<.........

FE073CBC: FE 00 14 84 FE 00 1D 3A FE 07 3C E8 FE 00 3E D4 .......:..<...>.

FE073CCC: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................

FE073CDC: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................

FE073CEC: FE 00 4C CA FE 07 3D 50 01 00 00 0C FE 07 65 E8 ..L...=P......e.

FE073CFC: 00 00 00 00 FE 07 3B A8 00 00 00 00 FE 00 71 80 ......;.......q.

FE073D0C: FE 07 3C 8C 00 00 00 00 00 00 00 00 00 00 00 00 ..<.............

FE073D1C: 00 00 00 00 00 14 00 00 1E FF 00 07 00 00 00 00 ................
```

```
FE073D2C: FE 07 3D 28 00 00 00 00 00 00 00 00 00 00 00 00 ..=(............
FE073D3C: 00 00 00 00 FE 00 1D 8C FE 00 09 2A 0E 00 08 07 ...........*....
FE073D4C: 00 00 00 00 FE 07 41 A0 01 00 00 8A 4F 56 46 4C ......A.....OVFL
FE073D5C: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
FE073D6C: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
FE073D7C: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
```

```
** Task #8 (TCB = FE073B48)
  Next Task (priority)...FE072690
  Next Task (timeout)....FE0715E0
  Stack Buffer...........FE072AF8
  Heap Buffer............FE0726F0
  Program Counter........FE007180
  Stack Pointer..........FE0739AE
  Code Segment..........00000000
  Data Segment...........00000000
  Start Parameters.......00000000
  Parent TID.............00000000
  Status Register........0008
  Page Latch............0000
  Priority...............0A
  Block Mode.............00
  Receive Mode...........01

  Task ID................0008
  Message Q Head.........FE077736
  Message Q Tail.........FE077736
  Blocked Timeout Value..00023BCC
  Blocked Message ID.....FFFFFFFF
  Blocked Message From...FFFFFFFF
```

```
 Blocked Message Code...0000FFFF

 D0 FE077698  D1 00023BCC  D2 00000000  D3 00000000
 D4 0E000201  D5 FE109CED  D6 0000000B  D7 0000000D
 A0 FE0741A8  A1 FE073B48  A2 00000000  A3 00000000
 A4 FE06FCF0  A5 FE001D3A  A6 FE073A12
 Running in " o".

 —Stack (TOS 100 bytes)
FE0739AE: FE 07 76 98 00 02 3B CC 00 00 00 00 00 00 00 00 ..v...;........
FE0739BE: 0E 00 02 01 FE 10 9C ED 00 00 00 0B 00 00 00 0D ................
FE0739CE: FE 07 41 A8 FE 07 3B 48 00 00 00 00 00 00 00 00 ..A...;H........
FE0739DE: FE 06 FC F0 FE 00 1D 3A FE 07 3A 12 FE 00 24 38 .......:...:...$8
FE0739EE: FF FF FF FF FF FF FF FF FF FF FF FF 00 00 00 13 ................
FE0739FE: FE 10 9C ED 0E 00 09 08 00 00 00 0D FE 07 77 E4 ..............w.
FE073A0E: FE 06 FC F0 FE 07 3B 38 FE 00 30 0E FE 00 0B F8 ......;8..0.....
FE073A1E: 0E 00 09 08 00 00 0D 64 0E 00 09 08 FF FF 80 66 .......d......f
FE073A2E: FF FF 80 00 FE 00 0B CE 0E 00 09 08 FE 07 77 E4 ..............w.
FE073A3E: FE 10 9C ED FE 10 9C ED 00 00 00 00 00 00 00 00 ................
FE073A4E: 00 00 00 00 FE 00 0B 9C 0E 00 09 08 FE 07 77 70 ..............wp
FE073A5E: FE 00 0B 6A 0E 00 09 08 00 00 0D 64 00 00 00 66 ...j.......d..f
FE073A6E: 00 00 04 78 FE 00 0B 3C 0E 00 09 08 FE 07 77 70 ...x...<......wp
FE073A7E: FE 10 9C ED 00 00 00 00 00 00 00 00 00 00 00 00 ................
FE073A8E: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
FE073A9E: 0E 00 09 08 00 00 00 00 FE 06 FF 60 FE 06 FB A0 ...........`....
      ** Message at FE077736
        Next.........00000000
        ID............FE0C7F44
        Code..........006B
        Status........0000
        Priority......0000
      From..........0E000201
      To............0E000908
      Sender Data...00000000
            00000000
            00000000
      Other Data....00000000
            00000000
            00000000
      Data Size.....00000000
      Data Pointer..00000000


** Task #9 (TCB = FE072690)
   Next Task (priority)...FE0715E0
   Next Task (timeout)....00000000
   Stack Buffer...........FE071640
   Heap Buffer............00000000
   Program Counter........FE007180
   Stack Pointer..........FE072624
   Code Segment...........00000000
   Data Segment...........00000000
   Start Parameters.......00000000
   Parent TID.............00000000
   Status Register........0004
   Page Latch.............0000
   Priority...............0A
   Block Mode.............00
   Receive Mode...........FF

   Task ID................0009
   Message Q Head.........00000000
```

```
  Message Q Tail.........FE0726C0
  Blocked Timeout Value..00000000
  Blocked Message ID.....00000000
  Blocked Message From...00000000
  Blocked Message Code...00000000

  D0 00000000  D1 00000000  D2 00000000  D3 00000000
  D4 00000000  D5 00000000  D6 00000000  D7 FE006183
  A0 FE0726C0  A1 FE072690  A2 00000000  A3 FE077736
  A4 00000000  A5 FE001D3A  A6 FE072680
  Running in " o".

  ---Stack (TOS 100 bytes)
FE072624: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
FE072634: 00 00 00 00 00 00 00 00 00 00 00 00 FE 00 61 83 ..............a.
FE072644: FE 07 26 C0 FE 07 26 90 00 00 00 00 FE 07 77 36 ..&...&.......w6
FE072654: 00 00 00 00 FE 00 1D 3A FE 07 26 80 FE 00 3B FA .......:..&...;.
FE072664: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
FE072674: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
FE072684: FE 00 4C CA FE 07 26 E8 01 00 00 0C FE 07 15 E0 ..L...&.........
FE072694: 00 00 00 00 FE 07 16 40 00 00 00 00 FE 00 71 80 .......@......q.
FE0726A4: FE 07 26 24 00 00 00 00 00 00 00 00 00 00 00 00 ..&$............
FE0726B4: 00 00 00 00 00 04 00 00 0A FF 00 09 00 00 00 00 ................
FE0726C4: FE 07 26 C0 00 00 00 00 00 00 00 00 00 00 00 00 ..&.............
FE0726D4: 00 00 00 00 FE 00 1D 7C FE 00 09 2A 0E 00 0A 09 .......|...*....
FE0726E4: 00 00 00 00 FE 07 2A F0 01 00 00 81 00 00 00 00 ......*.........
FE0726F4: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
FE072704: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
FE072714: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
```

```
** Task #A (TCB = FE0715E0)
 Next Task (priority)...FE073B48
 Next Task (timeout)....FE0741A8
 Stack Buffer...........FE070590
 Heap Buffer............00000000
 Program Counter........FE007180
 Stack Pointer..........FE07153E
 Code Segment..........00000000
 Data Segment..........00000000
 Start Parameters.......00000000
 Parent TID.............00000000
 Status Register........0008
 Page Latch............0000
 Priority...............0A
 Block Mode............01
 Receive Mode..........01

 Task ID................000A
 Message Q Head.........00000000
 Message Q Tail.........FE071610
 Blocked Timeout Value..00023BD6
  Blocked Message ID.....FFFFFFFF
  Blocked Message From...FFFFFFFF
 Blocked Message Code...0000FFFF

 D0 FE077698  D1 00023BD6  D2 00000000  D3 00000000
 D4 00023301  D5 0E000B0A  D6 0000001C  D7 0000C350
 A0 FE073B48  A1 FE0715E0  A2 00000000  A3 00000000
 A4 00000000  A5 FE001D3A  A6 FE0715A2
 Running in " o".
```

```
—Stack (TOS 100 bytes)
FE07153E: FE 07 76 98 00 02 3B D6 00 00 00 00 00 00 00 00 ..v...;.........
FE07154E: 00 02 33 01 0E 00 0B 0A 00 00 00 1C 00 00 C3 50 ..3...........P
FE07155E: FE 07 3B 48 FE 07 15 E0 00 00 00 00 00 00 00 00 ..;H...........
FE07156E: 00 00 00 00 FE 00 1D 3A FE 07 15 A2 FE 00 24 38 .......:......$8
FE07157E: FF FF FF FF FF FF FF FF FF FF FF FF 00 00 00 13 ................
FE07158E: 0E 00 0B 0A 0E 00 06 05 00 00 C3 50 FE 07 77 E4 ...........P..w.
FE07159E: 00 00 00 00 FE 07 15 D0 FE 00 3C 72 FE 00 13 2A ..........<r...*
FE0715AE: 0E 00 0B 0A 00 00 00 00 00 00 00 00 00 00 00 00 ................
FE0715BE: 00 00 00 00 00 00 00 00 00 00 00 00 00 02 2E 76 ..............v
FE0715CE: 00 0F 00 00 00 00 FE 00 4C CA FE 07 16 38 01 00 ........L....8..
FE0715DE: 00 0C FE 07 3B 48 FE 07 41 A8 FE 07 05 90 00 00 ....;H..A.......
FE0715EE: 00 00 FE 00 71 80 FE 07 15 3E 00 00 00 00 00 00 ....q....>......
FE0715FE: 00 00 00 00 00 00 00 00 00 00 00 08 00 00 0A 01 ................
FE07160E: 00 0A 00 00 00 00 FE 07 16 10 00 02 3B D6 FF FF ............;...
FE07161E: FF FF FF FF FF FF FF FF 01 01 FE 00 1D 84 FE 00 ................
FE07162E: 09 2A 0E 00 0B 0A 00 00 00 00 FE 07 26 88 01 00................
.*..........&...
```

The above sequence is repeated for each active Task Control Block.

In this example, the current task ID string shows the value of `0B000000`, which is the TID of the currently-running task. Refer to *Table 11-2* to determine the field name from which this value was obtained by checking the name of the string (`Current Task ID`), then looking in the structure indicated (`gCommon`) in the file listed (`arose.h`) to determine the actual location (`gTID`).

Table 11-2 shows a cross reference to the field names, listed in alphabetical order to the locations, structures, and include files in which the dumpcard fields are found.

▲ **Warning** These structures may change in future versions; therefore, do not hard code addresses for these locations. ▲

■ **Table 11-2** Dumpcard cross reference

Dumpcard field name	Location	Structure	Filename
Blocked Message Code	`pQMsgCode`	`pTaskSave`	`os.h`
Blocked Message From	`pQMsgFrom`	`pTaskSave`	`os.h`
Blocked Message ID	`pQMsgID`	`pTaskSave`	`os.h`
Blocked Timeout Value	`pQTimeout`	`pTaskSave`	`os.h`
CAP (Checksum)	`gCAP.iCksum`	`gCommon`	`arose.h`
CAP (Magic Number)	`gCAP.iMagic`	`gCommon`	`arose.h`
CAP (Pointer)	`gCAP.iPointer`	`gCommon`	`arose.h`
Code	`mCode`	`mMessage`	`os.h`
Code Segment	`pCodeSeg`	`pTaskSave`	`os.h`
Current Task ID	`gTID`	`gCommon`	`arose.h`
Current Task Pointer	`gCurrTask`	`gCommon`	`arose.h`
Data Pointer	`mDataPtr`	`mMessage`	`os.h`
Data Segment	`pDataSeg`	`pTaskSave`	`os.h`
Data Size	`mDataSize`	`mMessage`	`os.h`
Debugger Comm Area	`gDebugCom`	`gCommon`	`arose.h`
Debugger Flag	`gDebugOn`	`gCommon`	`arose.h`
Debugger Pointer	`gDebugTemp`	`gCommon`	`arose.h`
Error status	`gError`	`gCommon`	`arose.h`
Free message list	`gMsgFree`	`gCommon`	`arose.h`
FWA of initial code	`gFwaCode`	`gCommon`	`arose.h`
FWA of message area	`gFwaMess`	`gCommon`	`arose.h`
From	`mFrom`	`mMessage`	`os.h`
Heap Buffer	`pHeapBuf`	`pTaskSave`	`os.h`
ICC Task TID	`gIccTask`	`gCommon`	`arose.h`
ID	`mId`	`mMessage`	`os.h`

■ **Table 11-2** Dumpcard cross reference *continued*

Dumpcard field name	Location	Structure	Filename
Idle Chain	gIdleChain	gCommon	arose.h
Idle Loop Counter	gIdleLoop	gCommon	arose.h
Initial A5 value	gInitA5	gCommon	arose.h
Initial PC	gInitPC	gCommon	arose.h
LWA of initial code	gLwaCode	gCommon	arose.h
LWA of message area	gLwaMess	gCommon	arose.h
Major Cycles defered	gMajorDefer	gCommon	arose.h
Major Cycles skipped	gMajorSkip	gCommon	arose.h
Major Flag	gMajorFlag	gCommon	arose.h
Major Tick	gMajorTick	gCommon	arose.h
Memory buffer list ptr	gBuffList	gCommon	arose.h
Message Q Head	pQHead	pTaskSave	os.h
Message Q Tail	pQTail	pTaskSave	os.h
Messages discarded	gMsgBucket	gCommon	arose.h
Minor / Major Cycle	gMinPerMaj	gCommon	arose.h
Minor Tick Counter	gMinorTick	gCommon	arose.h
Name Task TID	gNameTask	gCommon	arose.h
Name Unregister pointer	gUnregTask	gCommon	arose.h
Next	mNext	mMessage	os.h
Next Task (priority)	pNextTask	pTaskSave	os.h
Next Task (timeout)	pNextTime	pTaskSave	os.h
Other Data	mOData[0] mOData[1] mOData[2]	mMessage mMessage mMessage	os.h os.h os.h
Page Latch	gPageLatch	gCommon	arose.h
Page Latch	pPageLatch	pTaskSave	os.h
Parent TID	pParentTID	pTaskSave	os.h
Priority	mPriority	mMessage	os.h

■ **Table 11-2** Dumpcard cross reference *continued*

Dumpcard field name	Location	Structure	Filename
Priority	pPriority	pTaskSave	os.h
Priority List	gPriList	gCommon	arose.h
Priority Table Pointer	gPriTable	gCommon	arose.h
Program Counter	pPcSave	pTaskSave	os.h
Release version	gRelease	gCommon	arose.h
Sender Data	mSData[0] mSData[1] mSData[2]	mMessage mMessage mMessage	os.h os.h os.h
Slot address	gSlotAdd	gCommon	arose.h
Slot number	gSlotNum	gCommon	arose.h
Stack Buffer	pStackBuf	pTaskSave	os.h
Stack Pointer	pSpSave	pTaskSave	os.h
Start Parameters	pStartParm	pTaskSave	os.h
Status	mStatus	mMessage	os.h
Status	pStatus	pTaskSave	os.h
Status Register	pSrSave	pTaskSave	os.h
Task ID	pTID	pTaskSave	os.h
Task Table Pointer	gTaskTable	gCommon	arose.h
Tick Chain	gTickChain	gCommon	arose.h
Ticks per Second	gTickPerSec	gCommon	arose.h
Time Base	gTimeBase	gCommon	arose.h
Timeout queue	gTimeout	gCommon	arose.h
Time Queue Flag	gTQFlag	gCommon	arose.h
Timer Task TID	gTimerTask	gCommon	arose.h
To	mTo	mMessage	os.h
Trace Task TID	gTraceTask	gCommon	arose.h
Unique Counter	gUnique	gCommon	arose.h

If A/ROSE crashes

The following steps will help you determine if A/ROSE has crashed and aid in finding the error:

1. First check location $64A to make sure that `Flag` is non-zero; that means A/ROSE has crashed.
2. If A/ROSE has crashed (`Flag` is non-zero), then examine `PC` located at location $642 and look at the address to which `PC` points.

◆ *Note:* The long word at location $650 is the exception number, which caused the 68000 to crash. This number is valid only if `Flag` at $64A is non-zero.

If A/ROSE does *not* detect an error, use the load map (described in the next section) to determine what code was executing at the time of the error. If A/ROSE *does* detect an error, it executes code that changes the error code to a symbolic name (described in the section following that).

Using the load map

You can use the load map produced when building your download file to examine locations on the card. Given a routine name in the load map, you can find where the routine actually exists on a smart card.

The starting addresses of the routines in A/ROSE in the load map are produced by the linker (refer to `makefile` in the file `MCP:Examples:A/ROSE:` for an example). The location `gCommon.gFwaCode` contains the first word address of the code containing the A/ROSE operating system that was downloaded to the card.

◆ *Note:* The location `0x000000` on the MCP card contains the initial stack pointer. The initial stack is at the high end of memory.

You can calculate the address of a routine within the load map in two ways. The first method calculates the code loaded that contains A/ROSE, as follows:

Address of routine on the card

= `gCommon.gFwaCode`

+ length of each previously loaded code segment

+ 4 * number of previously loaded code segments

The second method of calculation works whether the downloaded code contains the A/ROSE operating system or was dynamically downloaded, assuming that register A5 points to the Jump Table of the task. The second method calculates the code loaded to a smart card running A/ROSE, as follows:

Address of routine on the card

= value at location 4(A5)

+ 4

+ length of each previously-loaded code segment

+ 4 * number of previously-loaded code segments

This second method of calculation assumes that 4(A5) is approximately the address of the first code segment; this assumption may *not* always be the case.

Using A/ROSE error codes

If the A/ROSE operating system detects an error, it executes the following code (the PC at location $642 points to this code):

```
MOVE.L       #error code,gError

illegal
```

Within the common error-handling routines, A/ROSE changes the error code at location `gError` to the symbolic name `eBTHH`, and the processor executes a tight loop. *Table 11-3* lists the symbolic names of the error codes found in the files `:MR-DOS:includes:mrdos.a` and `:MR-DOS:includes:mrdos.h`.

■ **Table 11-3** Error codes for A/ROSE

Value	Symbolic Name	Location	Explanation
80000004	eBTHH	hwbuserr (osinit)	Bad things have happened
	or	hwerr (osinit)	Bad things have happened
$80000002	eCAIT	osinit	Cannot allocate idle task
$80000005	eCAMS	osinit	Cannot allocate message space
$80000006	eCAPR	osinit	Cannot allocate priority table
$80000001	eCAPT	osinit	Cannot allocate process table

continued...

■ **Table 11-3** Error codes for A/ROSE *continued*

Value	Symbolic Name	Location	Explanation
$80000009	eFMSG	tfreemsg (ostrap)	Attempt to free bad message buffer
$80000008	eMEMB	tfreemem (ostrap)	Attempt to free bad memory buffer
$80000003	eNPTR	pickTask (ostask)	No processes to run
$80000007	eOVFL	saveTask (ostask)	Stack overflow detected
$8000000A	eSMSG	tsend (ostrap)	Attempt to send bad message buffer
$8000000D	eSTPI	tstoptask (ostrap)	StopTask cannot be called from interrupt
$8000000C	eSTTI	tstarttask (ostrap)	StartTask cannot be called from interrupt
$8000000B	eTIMQ	tsend (ostrap)	Task not in timer queue

The symbolic names described below (listed in alphabetical order) match the error code returned, describe potential problems, and suggest how to find a solution.

eBTHH — Bad Things Have Happened

Description

Either a hardware error (hwerr) or hardware bus error (hwbuserr) has occurred.

Solution

When the A/ROSE operating system encounters an error, it executes the following code:

```
MOVE.L        #errorcode,gError
illegal
```

To check the hardware error, examine the PC at location $642 on the MCP card to see what code was being executed at the time of the problem. To check the hardware bus error, examine location $650 on the MCP card to see what hardware trap occurred.

◆ *Note:* Executing the illegal instruction causes the hwerr routine to be entered. The hwerr routine overwrites location gError with the error code eBTHH.

eCAIT — Cannot Allocate Idle Task

Description

The routine `osinit` executes an illegal instruction if it cannot allocate a Task Control Block for the Idle task.

The main routine calls the `osinit` routine to initialize A/ROSE. The routine `osinit` causes a crash if `osinit` cannot allocate enough memory for system data structure. This crash indicates a serious shortage of memory.

Solution

- Check the parameters sent to `osinit` in `osmain.c` (stack size and number of message buffers) and reduce as necessary.
- Make sure that the size of code is not too large for available memory. If necessary, rewrite to reduce the size of the code.
- Make sure that the initial stack pointer value in card location 0x0000 is valid. If invalid, download again. If still invalid after trying again, contact Apple Developer Services.

eCAMS — Cannot Allocate Message Space

Description

The routine `osinit` executes an illegal instruction if it cannot allocate the A/ROSE message buffer pool.

The main routine calls the `osinit` routine to initialize A/ROSE. The routine `osinit` crashes if it cannot allocate enough memory for system data structures. This crash indicates a serious shortage of memory.

Solution

- Check the parameters sent to `osinit` in `osmain.c` (stack size and number of message buffers) and reduce as necessary.
- Make sure that the size of code is not too large for available memory. If necessary, rewrite to reduce the size of the code.
- Make sure that the initial stack pointer value in card location 0x0000 is valid. If invalid, download again. If still invalid after trying again, contact Apple Developer Services.

eCAPR — Cannot Allocate Priority Table

Description

The routine `osinit` executes an illegal instruction if it cannot allocate the A/ROSE Priority Table.

The main routine calls the `osinit` routine to initialize A/ROSE. The routine `osinit crashes if it` cannot allocate enough memory for system data structures. This crash indicates a serious shortage of memory.

Solution

- Check the parameters sent to `osinit` in `osmain.c` (stack size and number of message buffers) and reduce as necessary.
- Make sure that the size of code is not too large for available memory. If necessary, rewrite to reduce the size of the code.
- Make sure that the initial stack pointer value in card location 0x0000 is valid. If invalid, download again. If still invalid after trying again, contact Apple Developer Services.

eCAPT — Cannot Allocate Process Table

Description

The routine `osinit` executes an illegal instruction if it cannot allocate the A/ROSE process table.

The main routine calls the `osinit` routine to initialize A/ROSE. The routine `osinit` crashes if it cannot allocate enough memory for system data structures. Any of these crashes indicates a serious shortage of memory.

Solution

- Check the parameters sent to `osinit` in `osmain.c` (stack size and number of message buffers) and reduce as necessary.
- Make sure that the size of code is not too large for available memory. If necessary, rewrite to reduce the size of the code.
- Make sure that the initial stack pointer value in card location 0x0000 is valid. If invalid, download again. If still invalid after trying again, contact Apple Developer Services.

eFMSG — Attempt to Free Bad Message

Description

The routine `tfreemsg` is a kernel trap routine that performs the work of a `FreeMsg` request. The `tfreemsg` routine executes an illegal instruction if it determines that the pointer to the message it is attempting to free is invalid or the message is not in use.

Solution

Verify that the pointer passed to `FreeMsg` points to a valid, in-use message buffer:

The message buffer is preceded by a four-byte header indicating whether the message buffer is in use or available. The first three bytes contain the characters `MSG`. The fourth byte contains one of the following:

- `0xFs` (where *s* is the slot number) if the message buffer is in use
- `0x20` (space) if the message buffer has never been used
- `0x00` if the message buffer has been used but is now available for reuse
- `0xFF` if ICCM has obtained a message for internal use
- `0xsF` (where *s* is the slot number) if message is on the internal A/ROSE queue

◆ *Note:* If the fourth byte is `0x00`, the application code may be attempting to free a particular message multiple times.

Diagnose and correct the user code.

eMEMB — Attempt to Free Bad Memory Buffer

Description

The `tfreemem` routine is a kernel trap routine that performs the work of a `FreeMem` request. The `tfreemem` routine executes an illegal instruction if it is invoked with a bad memory buffer pointer; that is, the pointer does not point to an area of memory that was allocated by a previous `GetMem` request or an attempt was made to free a memory buffer that was already freed.

Solution

Check the pointer passed to the `FreeMem` request. Verify that it points to a valid memory buffer.

The buffer address must be equal to or greater than the address stored in `gCommon.gBuffList`. The eight bytes in front of the buffer area pointed to should contain a memory buffer header of the form:

```
bHeader                 RECORD 0

bNext     DS.L 1        ; pointer to next header (32-bit NuBus form)

bUsage  DS.B 1          ; usage count : 0=free; nonzero=allocated

bSize     DS.B 3        ; Size of block in 8-byte chunks

ENDR
```

If the buffer header is invalid, determine where in the user code the buffer header has been corrupted and correct the code. If the buffer header appears to be valid, the buffer pool links may be corrupted. Verify the buffer pool links as follows:

1. Get the pointer to the buffer pool area (`gCommon.gBuffList`). This points to the first buffer header (`bHeader`).
2. Get the pointer to the next memory block header from `bHeader.bNext`. This pointer can also be determined by the following equation (except for the last buffer, which has a `bHeader.bNext` pointer of zero):

   ```
   bHeader.bNext = bHeader.bSize * 8 + bHeader
   ```

   If `bHeader.bNext` does not equal the result of the calculation, a buffer header has been corrupted.

   Check `bHeader.bUsage` to determine if the buffer is free or allocated (see header). There should not be multiple free adjacent buffers.
3. Repeat 2 until `bHeader.bNext` is zero, indicating that this is the last buffer, or a buffer pool corruption is discovered.

If a buffer pool corruption has caused the crash, diagnose and correct the user code that caused the corruption. Otherwise, call Apple Developer Services.

eNPTR — No Processes to Run

Description

The `pickTask` routine chooses the next task to schedule when the current task gives up the CPU. The `pickTask` routine executes an illegal instruction if there is no task available for execution. The Idle task should *always* be available for execution.

Solution

- Ensure that the Idle task is available for execution (no routine on the Idle Chain should invoke a `Receive` request). If a routine on the Idle Chain invokes `Receive`, correct the code.

  The MPW tool dumpcard can be used to determine the state of the Idle task at the time of the crash.

  1. Run `dumpcard -e -r` under MPW.
  2. Locate the task control block for Task 0.
  3. Check the `Status` line. If the word `(Blocked)` appears, the Idle task is blocked from execution by a `Receive` request.
- If the Idle task appears to be available for execution, call Apple Developer Services for help.

eOVFL — Stack Overflow Detected

Description

The saveTask routine stores the context of the current task when the current task gives up the CPU. The saveTask routine executes an illegal instruction if it detects an apparent overflow condition in the user's stack area.

The system inserts the string OVFL at the end of the user stack at user task startup time. The task saveTask checks for this string to determine if the user stack is corrupted; pStackBuf within pTaskSave points to the OVFL string. The pStackBuf pointer is a single element within pTaskSave, located in the file os.a and the file os.h.

Corruption of A/ROSE global data structures can also cause this crash.

Solution

Correct the user code that causes the stack overflow. If a condition that causes a stack overflow cannot be found, use the MPW dumpcard tool to display the A/ROSE data structures and investigate for inconsistencies.

Verify that:

- Current Task ID (gCommon.gTID) is valid
- Current Task Pointer (gCommon.gCurrTask) points to the Task Control Block (TCB) of the currently-executing task
- TCB of current task is valid

eSMSG — Attempt to Send Bad Message Buffer

Description

The tsend routine is a kernel trap routine that performs the work of a Send request. If tsend determines that the pointer to the message it is attempting to send is invalid or the message is not in use, tsend executes an illegal instruction and causes a crash.

Solution

Verify that the pointer passed to Send points to a valid, in-use message buffer (refer to eFMSG for the crash solution). Diagnose and correct the user code.

eSTPI — Stop Task cannot be called from interrupt routine

Description

The tstoptask routine checks that StopTask is not called from an interrupt routine.

Solution

Correct the code that issued the StopTask request.

eSTTI — Start Task cannot be called from interrupt routine

Description

The `tstarttask` routine checks that `StartTask` is not called from an interrupt routine.

Solution

Correct the code that issued the `StartTask` request.

eTIMQ — Task Not in Timer Queue

Description

The `tsend` routine is a kernel trap routine that performs the work of a `Send` request. If the message being sent can satisfy an outstanding `Receive` request, or the `Receive` request has specified a timeout value but `tsend` could not locate the receiving task in the A/ROSE timeout queue, then `tsend` executes an illegal instruction. This crash indicates a corruption of the timeout queue.

Solution

Use the MPW dumpcard tool to display the A/ROSE data structures and investigate for A/ROSE timeout queue corruption:

1. Execute `dumpcard -e -m -r` from MPW.
2. Get the address of timeout queue. This is the address of the first Task Control Block (TCB) on the timeout queue.
3. Search for a TCB that is located at the timeout queue address
4. Get the address of the next TCB in the queue from the `NextTask (timeout)` entry in the TCB.
5. Search for the next TCB in the queue.
6. Repeat 4 and 5 until either `NextTask (timeout)` is `00000000`, indicating the end of the timeout queue chain, or it points to a value that is not a valid TCB, indicating a corruption of the timeout queue.

If the timeout queue is corrupted, locate the code that caused the corruption, and fix it.

Task Not Stopped

There is no associated error code for this problem.

Description

The `deadMan` routine executes when a user task runs to completion. `deadMan` executes an illegal instruction if it cannot stop the task that has just completed. This crash indicates a problem in the A/ROSE kernel. The global area `gCommon` may have been corrupted or the task control block for a particular user task may have been corrupted.

Solution

If user code has not corrupted memory, call Apple Developer Services.

If A/ROSE hangs

If your system appears to be nonfunctional but you have determined that your code has not crashed, then your system may be hung; that is, the CPU may still execute instructions, but the section of code being executed will never give up control of the CPU.

Determining the cause of a hang can be a difficult process. This section provides information and guidelines to use in investigating hangs. It does not cover all possible problems.

When the system is healthy, a hardware timer routinely activates a timer interrupt routine. The interrupt routine decrements a counter, `gCommon.gMinorTick`, each time the routine is executed. Every ***nth*** time the timer interrupt routine executes (where ***n*** is a configuration parameter), the interrupt routine increments a counter, `gCommon.gMajorTick`. Thus, `gCommon.gMajorTick` increments every ***nth*** decrement of `gCommon.gMinorTick`.

During major tick processing (whenever `gCommon.gMajorTick` increments), the timer interrupt routine performs the following:

- sets `gCommon.gMajorFlag` to non-zero to indicate major tick processing in progress
- resets `gCommon.gMinorTick`
- executes any routines on the Tick Chain are executed
- if the current task is running in slice mode, the system task scheduling mechanism schedules a new task
- sets `gMajorFlag` to zero to indicate the end of major tick processing

The health of the major and minor tick counters provides an indication of the state of the system. First, examine the `gMajorTick` counter: if it is incrementing, go to the section titled "gMajorTick is incrementing;" otherwise, go to the section titled "gMajorTick is not incrementing."

gMajorTick is not incrementing

The following sections are provided to help the user diagnose why the `gMajorTick` counter is not incrementing.

Any of the following events could stop the timer routine from incrementing `gCommon.gMajorTick` (assuming the system has not crashed). These events are referred to by letters for later reference to each event.

A. A piece of code disables interrupts and goes into an infinite loop (never exits).

B. Interrupt code servicing an interrupt of higher priority than the timer interrupt goes into an infinite loop.

C. Interrupt code servicing an interrupt of higher priority than the timer interrupt may not properly clear the interrupt, which then appears as a continuously-generated interrupt.

D. A routine on the Tick Chain is infinitely looping.

E. A routine on the Tick Chain corrupts the system stack during tick chain processing.

F. A programmable hardware timer may have been improperly set up or accidentally changed.

Determining the cause

To identify the cause of the problem, follow the steps listed below to determine why `gMajorTick` is not incrementing:

1. Examine the `gCommon.gMinorTick` counter to see if it is changing.
   - □ If it is not changing, the problem could be (A), (B), (C) or (F) of the above list.
   - □ If it is changing, the problem could be (D) or (E).
2. Try each interrupt level once. Starting at the lowest interrupt level , change each interrupt autovector to the value `0x00000001`.

   If an interrupt is continuously generated, an address error exception occurs at the address of the autovector associated with that interrupt. Examine the 0x0600 area to see if an address error exception occurred.

   The problem is (C) above if the code to clear the interrupt is wrong. Examine the code to determine its correctness.

   The problem is (B) above if the code to process an interrupt of lower priority than the interrupt priority executing at the time the machine hangs is infinitely looping (that is, never executing an RTE). Examine the supervisor stack for other interrupt routine addresses to determine if other interrupt code is currently being processed.

▲ **Warning** Be sure to save your files before trying the next step; the 68000 processor on the smart card may crash the Macintosh II computer during this operation. ▲

4. If all else fails, execute the MPW dumpcard tool to halt the smart card.
   - □ Type `dumpcard -h` to halt the 68000 processor on the smart card.
   - □ Examine the `PC` stored in area 0x0600 to determine what code was being executed at the time of the halt.
   - □ Examine the `SR` stored in area 0x0600 to determine what interrupt level and state (user/supervisor) the 68000 processor was in .
   - □ Examine the appropriate stack (user/supervisor) and analyze the information found to determine the cause of the hang.

gMajorTick is incrementing

If `gCommon.gMajorTick` is incrementing properly, the A/ROSE kernel is not hung; that is, everything appears healthy from the point of view of the operating system. However, one or more user tasks may be hung. Use the MPW dumpcard tool to examine the state of the hung system.

The following events that can cause one or more tasks to appear hung. These events are listed in the order of greatest probability of happening; check each cause in turn.

◆ *Note:* This list is not complete, and cannot be. The events listed here are provided as guidelines for commonly found problems.

1. A task has invoked a blocking `Receive` request for a message, but never receives a message to satisfy the request; the task is never rescheduled for execution.
2. A/ROSE runs out of message buffers and a task loops on a `GetMsg` call, waiting for a buffer.
3. A task may be running continuously in block mode without executing a blocking `Receive` or a `Reschedule`; other tasks never get a chance to execute.
4. A task of high priority may be running in slice mode and not doing a blocking `Receive` to relinquish the CPU; lower-priority tasks running in slice mode never have a chance to execute.
5. Code on the Idle Chain may be executing in an infinite loop.

Each of these are described more fully in this section.

Is a task waiting on a blocking Receive request?

If tasks appear to be behaving properly, but the task just refuses to do anything useful, check the following: Is this task doing a blocking `Receive` request with bad matching criteria? Is the task waiting for a reply that will never be received?

- Examine the `Current Task ID` to get the number of the currently-executing task.
- `pTasksave.`prcvFlag is non-zero if the task is doing a blocking `Receive` request.
- Examine `pTaskSave.pPcSave`. This is the `PC` where the task will begin execution when its blocking `Receive` request is satisfied. The `PC` may be in the `Receive` code in the glue library. The stack for this task should also be examined to determine what routine the `Receive` code in the glue library will return to.
- Examine `pTaskSave.pSpSave`. This is the saved user stack pointer. The user stack has the following format when the task is not currently executing:

  The top of the stack contains the registers for this task in the following order: `D0-D7, A0-A6,` followed by the rest of the stack.

- Examine the code that the task is currently executing to determine what message the task should be waiting for.
- `pTaskSave.pRcvFlag` is greater than zero if the task is waiting for a message with specific matching criteria.
- Examine the `pTaskSave.pqMsgID`, `pTaskSave.pQMsgFrom`, and `pTaskSave.pQMsgCode` fields. These fields are the specific matching criteria fields . Ensure that the matching criteria makes sense given what message the task should be waiting for.
- Examine `pTaskSave.pQHead`. This pointer points at the first message buffer on the task's `Receive` message queue. The pointer is zero if no message is waiting on the task's `Receive` message queue.
- Examine any waiting messages for this task. Determine if any waiting message is the message that the task should be waiting for. Determine if any waiting message signifies an error condition that indicates that the task will not receive the message it is waiting for.
- If the task is waiting for a message that is from a task on another card, ensure that the other card has not crashed or hung. Ensure that the other card has enough message buffers. Ensure that the sending task on the other card is not itself hung.
- If the task is waiting for a message that is from a task on another card, attempt to determine if intercard communication between the two cards is occurring.

The symbol `gCommon.gCAP.CaPtr` points at the local intercard communications area on this card. Use this pointer to find the intercard communications area. The files `iccmDefs.a` and `:A/ROSE:INCLUDES` describe the intercard communications area. These files are for debugging purposes only.

The structure `ca_Rec` is the intercard communications area.

The arrays `ca_Rec.IFlags`, `ca_Rec.Addrs`, and `ca_Rec.Ptrs` are indexed by the slot number of the card. The Macintosh II is treated as slot 0.

`ca_Rec.Addrs` is an array of pointers to other intercard communications areas that the local card knows about. Make sure that both the local and the remote intercard communications areas know about each other. Intercard communications have been lost should the respective `ca_Rec.Addrs` field in either card be zero.

`ca_Rec.Ptrs` is an array of pointers to message buffers. When two cards are communicating, the respective `ca_Rec.Ptrs` should contain the addresses of message buffers. If they do not, a card may have run out of message buffers.

Has A/ROSE run out of message buffers?

Check the pointer `gCommon.gMsgFree`. This pointer is zero if no free message buffer is available. Tasks cannot communicate with each other if A/ROSE runs out of message buffers.

The following are possible causes for running out of message buffers:

- An insufficient number of message buffers may have been specified as a parameter to `osinit`.
- The message buffer free list pointed to by `gCommon.gMsgFree` has been corrupted .
- A task is allocating message buffers but not freeing them with `FreeMsg`.
- The message buffers are accumulating on a task's `Receive` message queue and not being processed by the task.

The following should be checked to determine the cause (see the previous description of a message header):

- The symbol `gCommon.gFwaMess` points at the first byte of the message buffer area. Look and see what is in the message buffers that are in use.
- Check the header of each message buffer to see if any are free. Any free message buffer should be linked to the `gCommon.gMsgFree` message buffer list.
- Look at the task control blocks of the tasks to see if any task has a large number of message buffers on its `Receive` message queue.

Is a task running in block scheduling mode?

A task running in block scheduling mode must periodically do either a blocking `Receive` request or a `Reschedule` request to let other tasks execute. A blocking `Receive` request is a request with a positive or zero timeout value.

No other task will be able to run if a task running in block scheduling mode does not do a blocking `Receive` request or a `Reschedule` request.

In particular, the ICCM is responsible for forwarding messages to other cards. It runs as a user task and will never execute if a task runs in block mode and never executes a blocking `Receive` request or a `Reschedule` request.

- Determine which task is currently executing.
- Examine its code to ensure that it is periodically doing either a blocking `Receive` or a `Reschedule` request to allow other tasks to execute.

Is a task executing in an infinite loop in slice scheduling mode?

A task running in slice mode must periodically execute a blocking `Receive` request to allow lower-priority tasks to be scheduled for execution. Tasks of equal or higher priority than the infinitely looping task will continue to run. Tasks of lower priority will not execute.

Determine what tasks are currently executing. Examine the code for the currently executing tasks to ensure that they are periodically doing a blocking `Receive` request to allow the scheduling lower-priority tasks .

Is code on the Idle Chain executing in an infinite loop?

The Idle task executes the Idle Chain while in block scheduling mode.

- Determine which task is currently executing. If it is the Idle task, examine the code on the Idle Chain to ensure that the code is not executing in an infinite loop.

▲ **Warning** Be sure to save your files before trying the following step; the 68000 processor on the smart card may crash the Macintosh II computer during this operation. ▲

- From MPW, type `dumpcard -h` to try to halt the 68000 processor on the card.
- Once halted, examine the PC stored in area 0x0600 to determine the code that was being executed.

Troubleshooting A/ROSE Prep

This section describes the following events that can occur during A/ROSE Prep processing:

- illegal instructions
- `DebugStr` a-line trap calls that are executed
- hang conditions

To assist in troubleshooting during your development efforts, both error codes and error messages have been integrated into the code for the A/ROSE Prep driver. Error messages are displayed on the screen when you use a debugger; error codes are not displayed. A positive number indicates a message pointer; zero indicates no message or error code; and a negative number indicates an error code.

Two types of errors can occur when calling the A/ROSE Prep driver. The first type is more informative and provides error codes or messages (listed in the following tables). When the A/ROSE Prep driver detects a serious error, however, it executes the following instructions:

```
PEA          MsgAddress    ; Address of error message

_DebugStr                  ; Call Debug A-Trap
```

Table 11-4 lists the A/ROSE Prep driver error codes returned from an A/ROSE Prep `Receive` request. Each of these codes is described, along with a potential solution.

■ **Table 11-4** Error codes for A/ROSE Prep driver

Value	Name	Explanation
-64	NoQueueErr	No more queues available
-65	QueueBusy	Receive already outstanding on queue

Table 11-5 lists the possible error messages from the INIT resource that installs the A/ROSE Prep driver. Each of these messages is described in this section, along with a potential solution.

■ **Table 11-5** Error messages from the INIT resource

Error message string
A/ROSE Prep INIT31 - Unit Table full
A/ROSE Prep INIT31 - No DRVR resource in file
A/ROSE Prep INIT31 - Failed to open driver

Table 11-6 lists the possible error messages from the A/ROSE Prep driver or Name Manager. Each of these messages is described in this section, along with a potential solution.

■ **Table 11-6** Error messages from the A/ROSE Prep driver/Name Manager

Error message string
A/ROSE Prep Freemsg – Bad message pointer
A/ROSE Prep Send – Bad message pointer or mFrom
A/ROSE Prep – Missing resource: A/ROSE Prep Entries
A/ROSE Prep – Unable to get space from system heap
A/ROSE Prep Name Manager – Missing aipn resource: NameManagerEntries
A/ROSE Prep KillReceive/CloseQueue – timeout queue error
A/ROSE Prep Send – timeout queue error
A/ROSE Prep Periodic processing – timeout queue error
A/ROSE Prep Receive – timeout queue error
A/ROSE Prep Receive – Interrupt routine did blocking Receive

If A/ROSE Prep crashes

This section describes the crashes that can occur with A/ROSE Prep because of improper parameter usage; corruption of either the A/ROSE Prepdriver or its internal data structures; corruption of the A/ROSE Prepinternal data structures during request execution or periodic processing; or during invocation of the A/ROSE Prepdriver or the A/ROSE Prep Name Manager.

A/ROSE Prep crashes during Macintosh startup

During Macintosh II startup, an INIT31 resource found in the A/ROSE Prep file installs the A/ROSE Prep driver and the A/ROSE Prep Name Manager. The INIT31 resource may crash by executing a `DebugStr` call if it detects a serious problem. These potential problems are described in the following sections.

`A/ROSE Prep INIT31 - Unit Table full`

INIT31 executes a `DebugStr` call if there is no empty slot in the driver Unit Table pointed to by `UTableBase`, indicating that there are too many drivers configured in the Macintosh II system. (Refer to *Inside Macintosh* for more information on the Unit Table.)

Solution

Boot from another system disk and either remove the A/ROSE Prep file or remove another driver.

`A/ROSE Prep INIT31 - No DRVR resource in file`

INIT31 executes a `DebugStr` call if it does not find a driver of resource type `'DRVR'` and resource name `'.IPC'` in the A/ROSE Prep file. This indicates that the A/ROSE Prep file is in error (due to improper file generation or data corruption).

Solution

Boot from another system disk and replace the A/ROSE Prep file.

```
A/ROSE Prep INIT31 - Failed to open driver
```

INIT31 executes a `DebugStr` call if the A/ROSE Prep driver cannot be opened successfully. This indicates there is a serious problem either with the A/ROSE Prep driver or with the Macintosh II operating system.

Solution

Boot from another system disk and replace the A/ROSE Prep file.

A/ROSE Prep crashes with improper parameter usage

This section describes the events relating to improper parameter usage that can cause the A/ROSE Prep driver to crash. These crashes occur when the A/ROSE Prep driver detects a bad message pointer passed as a parameter to a driver request.

The A/ROSE Prep driver considers a message buffer pointer to be bad if it either does not point to a message buffer or the message buffer pointed to is not in use.

Every message buffer is preceded by a four-byte header, indicating whether the message buffer is in use or available. The first three bytes are the characters `MSG`. The fourth byte is one of the following:

- `0x20` (a space) if the message buffer has never been used
- `0x00` if the message buffer has been used but is now available for re-use
- `0xF0` if the message buffer is in use
- `0x0F` if the message is currently on an internal A/ROSE Prep queue
- `0xFF` if ICCM has obtained a message for internal use

A/ROSE Prep FreeMsg — Bad message pointer

Description

The A/ROSE Prep driver executes a `DebugStr` call if user code invokes a `FreeMsg` request with a bad message pointer.

Solution

Diagnose problem, correct code, and retry.

`A/ROSE Prep Send - Bad message pointer or mFrom`

The A/ROSE Prep driver executes a `DebugStr` call if user code invokes a `Send` request with a bad message pointer.

Solution

Diagnose problem, correct code, and retry.

A/ROSE Prep crashes during driver initialization

The A/ROSE Prep driver and the A/ROSE Prep Name Manager can cause a crash due to detection of data corruption during their initialization sequences.

`A/ROSE Prep - Missing resource: A/ROSE Prep entries`

The A/ROSE Prep driver executes a `DebugStr` call if it does not find a resource of type `'aipn'` and name `'A/ROSE Prep Entries'` in the A/ROSE Prep file, indicating that the A/ROSE Prep file was either improperly generated or corrupted after generation.

Solution

Boot from another system disk and either replace the `A/ROSE Prep` file or add the missing resource to the `A/ROSE Prep` file using ResEdit. The resource consists of two 16-bit words as follows:

- The first 16-bit word is the stack size in bytes of the stack to be used when completion routines are called (initial value is 0x1000).
- The second 16-bit word is the number of message buffers to be permanently allocated (initial value is 0x0064).

`A/ROSE Prep - Unable to get space from system heap`

At startup, the A/ROSE Prep driver executes a `DebugStr` call if it cannot allocate a 12-byte nonrelocatable block from the system heap. Either the Macintosh has insufficient memory or used all of the available system heap.

Solution

This crash indicates a serious system problem (such as configured in the System Folder or in the System file). Diagnose and correct the problem and retry. You may need to reduce the number of applications, or remove or fix the driver or Init31 resource.

A/ROSE Prep Name Manager — Missing aipn resource: Name Manager entries

The A/ROSE Prep Name Manager executes a `DebugStr` call if it does not find a resource of type `'aipn'` and name `'NameManagerEntries'` in the `A/ROSE Prep` file, indicating that the A/ROSE Prep file was either improperly generated or corrupted after generation.

The Pascal string `'NameManagerEntries'` immediately follows the illegal instruction. This string can be used to verify that the crash is, in fact, the Name Manager `'Missing Resource'` crash.

Solution

To correct the problem, add the missing resource to the `A/ROSE Prep` file using ResEdit. The resource consists of a 16-bit word indicating the number of entries allowed in the Name Managers tables (the initial value is `0x0012`).

IPC driver crashes during execution

The following events cause a crash if the A/ROSE Prep driver detects corruption of its internal data structures during request execution or periodic processing:

- invocation of a `KillReceive` or a `CloseQueue` request
- receipt of a message that satisfies a previous `Send` or `Receive` with timeout request
- a `Receive` request with a positive timeout
- interrupt routine did a blocking `Receive`

A/ROSE Prep KillReceive/CloseQueue — timeout queue error

The A/ROSE Prep driver executes a `DebugStr` call if there is an a outstanding `'Receive with timeout'` request and either a `KillReceive` or a `CloseQueue` request is invoked, but the driver detects internal data corruption during processing of the request.

Solution

Report the problem to Apple Developer Services.

`A/ROSE Prep Send – timeout queue error`

There may be situation in which a `'Receive with timeout'` request is outstanding and a message that satisfies the `Send` request becomes available. If the driver detects internal data corruption during processing of the `Send` request, the A/ROSE Prep driver executes a `DeBugStr` call.

Solution

Report the problem to Apple Developer Services.

`A/ROSE Prep Periodic processing – timeout queue error`

The A/ROSE Prep driver executes a `DebugStr` call if a `'Receive with timeout'` request times out, but the driver detects internal data corruption during processing of the timeout.

Following this call to `DebugStr`, the A/ROSE Prep driver immediately and unconditionally branches back to the code that called `DebugStr`. This can be used to verify that the crash is, in fact, a timed-out `Receive` request crash.

Solution

Report the problem to Apple Developer Services.

`A/ROSE Prep Receive – timeout queue error`

The A/ROSE Prep driver executes a `DebugStr` call if a `'Receive with timeout'` request is outstanding and a message that satisfies the `Receive` request becomes available, but the driver detects internal data corruption during processing of the message.

Solution

Report the problem to Apple Developer Services.

`A/ROSE Prep Receive – Interrupt routine did blocking Receive`

The A/ROSE Prep driver executes a `DebugStr` call if it detects an interrupt routine that does a blocking `Receive` request. (Interrupt routines may not do a blocking `Receive` request.)

Solution

Change your interrupt routine.

IPC Name Manager crashes during execution

The Name Manager executes an illegal instruction when it detects an internal problem. These do not call the `DebugStr` routine.

Name Manager Receive with Completion

If the A/ROSE Prep Name Manager issues a `Receive` request with a completion routine specified and the request fails, the Name Manager executes an illegal instruction.

Solution

Report the problem to Apple Developer Services.

Name Manager Receive Request Failure

If the A/ROSE Prep driver invokes a Name Manager `Receive` request with a completion routine specified and provides an error indication instead of a valid message buffer pointer, the Name Manager executes an illegal instruction.

Solution

Report the problem to Apple Developer Services.

Name Manager Receive Request without Completion

Tf the A/ROSE Prep Name Manager issues a nonblocking `Receive request` with no completion routine specified and the request returns an error indication instead of a valid message buffer pointer, the Name Manager executes an illegal instruction.

Solution

Report the problem to Apple Developer Services.

If the IPC glue code crashes

This section describes troubleshooting guidelines for Macintosh II applications using the A/ROSE Prep driver.

Requests to the A/ROSE Prep driver are made through a glue library. The glue library provides an interface between the calling code and driver code, allowing future driver changes to be made transparent to the user.

The glue library initializes on invocation of the first driver request (with a command such as `GetMsg`, `Send`, `Receive`, `GetTId`, and so forth). The glue library executes an illegal instruction if the A/ROSE Prep driver could not be opened successfully, or the glue library could not be properly initialized.

A glue library illegal crash is surrounded on either or both sides by multiple instances of the following instructions:

```
        LEA     LocBlock, A0

        JSR     SetJmpT

        illegal
```

△ **Important** For crashes of this type, you must report the problem to Apple Developer Services. △

Detection of these instructions can be used to verify that the crash is, in fact, a glue library crash.

If A/ROSE Prep hangs

The Macintosh II may appear to be hung while executing A/ROSE Prep request code. This section describes some of the conditions under which your system will appear to be hung and provides suggested solutions to these problems.

Events that cause A/ROSE Prep to hang

The following sections describe two of the most common events that could result in this condition.

Macintosh II 32-bit mode debugger hangs

Description

The A/ROSE Prep driver accesses the smart card's memory in 32-bit memory mode. Older versions of some debuggers cannot handle bus errors; if the Macintosh II is running in 32-bit mode, the debugger can freeze the Macintosh II.

The following events can cause a hang while the Macintosh II is in 32-bit mode:

- Invoking `CopyNuBus` with invalid source or destination addresses.
- A smart card hardware problem resulting in a bus error.

Solution

Be sure that your debugger can handle 32-bit mode, and reboot.

blocking Receive Request is Unsatisfied

Description

The Macintosh II will appear to hang if a task issues a blocking `Receive` request for which no message is available.

Solution

The following are two possible solutions:

1. The A/ROSE Prep driver periodically calls the routine specified in the `OpenQueue` request while processing a blocking `Receive` request. This routine could issue a `KillReceive` or `CloseQueue` request to cancel the blocking `Receive` request.
2. A positive timeout value can be specified for the blocking `Receive` request. The A/ROSE Prep driver returns a zero message pointer should the time specified elapse.

Examining the A/ROSE Prep global area

You can examine the A/ROSE Prep global area to determine the state of a task on the Macintosh II that is using the A/ROSE Prep driver. You'll want to examine the global area when the Macintosh II does not appear to be hung and the task appears to be doing nothing (rather than what it's supposed to be doing). `IPCg` is the global area, described in the includes files `IPCgDefs.a` or `IPCgDefs.h` in the A/ROSE Prep folder.

Examine the A/ROSE Prep global area to determine if there

- is a task waiting for a message
- are any matching criteria that must be met for the task to receive a message
- are any messages currently queued waiting to be received by the task
- are any free message buffers

There are many ways to find the A/ROSE Prep global area; these methods get progressively more complicated, but yield the same results. The most common method is the most simple and easiest to use. Simply issue a `GetIPCg` request to the A/ROSE Prep driver. The driver returns the starting address of the A/ROSE Prep global area.

Part III **Hardware Development**

Part III, Hardware Development,provides:

- MCP card description and specifications
- overview of NuBus on the MCP card, with functional examples
- PAL listings for the MCP card

Chapter 12 MCP Card Specifications

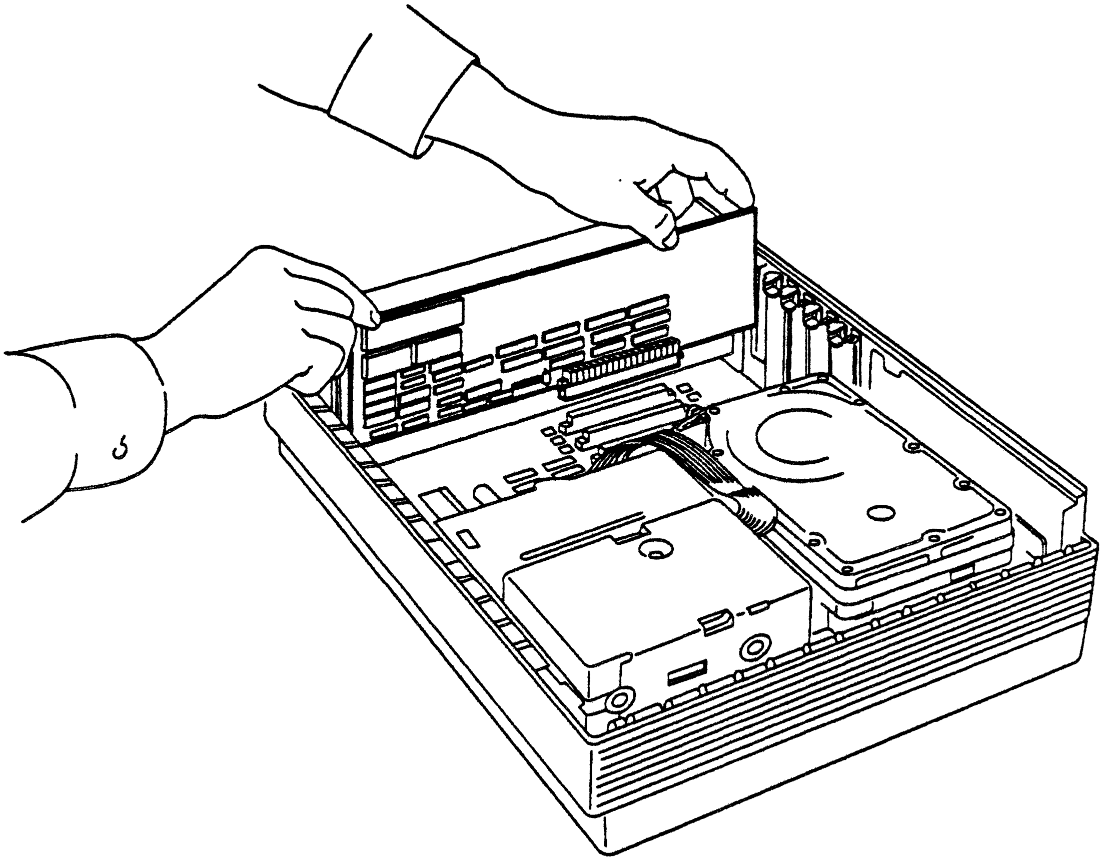
THIS CHAPTER describes the hardware portion of the Macintosh Coprocessor Platform and provides descriptions of the components of the MCP card. ■

Introduction to the MCP card

The MCP card is a generic master/slave I/O processor card. This smart card has a full NuBus master/slave interface with a 68000 processor on board. The 68000 can access any device on NuBus, and the memory and I/O of the 68000 can be accessed by any device on NuBus.

Figure 12-1 shows the MCP card being installed in a Macintosh II computer.

- **Figure 12-1** MCP card installed in the Macintosh II



MCP card description

There are approximately 26 square inches of prototyping space on a standard size MCP card. This area is provided for developing an interface logic to connect to the communications link of the developer's choice.

Electrically, the interface is the 16-bit 68000 processor bus. The added interface logic should decode the 4000-9FFF address space for all accesses. Refer to the information in the section describing the address map for additional details.

This section provides detailed descriptions of the following elements of the MCP card:

- The functional components, including processor, ROM, and RAM
- Address map
- Timer
- Reset
- Interrupts
- NuBus interface
- ASIC MCP Support

ASIC MCP support

ASIC MCP refers to the version of Macintosh Coprocessor Platform NuBus card implemented using the Application Specific Integrated Circuits (ASICs.) The ASICs used in the ASIC MCP cards implement all the NuBus bus protocols. There are a few differences between a regular MCP card and an ASIC MCP card in terms of controlling the board (card.) Also, there is a programmable timer in the ASCI MCP card whereas the regular MCP card has a fixed interval timer. A/ROSE 1.1 supports the ASIC MCP card. ASCI MCP cards have unique board IDs. The new download routines use BoardIDs to select the correct card dependent routines to use for downloading to a particular ASIC MCP Card.

Processor

The I/O Processor utilizes a 10 Megahertz (MHz) 68000 processor with no wait states for access to onboard RAM. The 10 MHz clock is derived from the 10 MHz NuBus clock. All access by the 68000 is implemented by a 16-bit data bus, with byte mode also supported.

ROM

The 16-bit-wide ROM is implemented with two 256-Kilobit (Kbit) ROMS, yielding a 64-Kilobyte (KB) ROM space. The ROM:

- serves as power-up code for the 68000
- provides a place for user firmware
- stores the NuBus ID data for the card

The ROM inserts one wait state when accessed by the on-board 68000. To the NuBus interface, ROM appears as a full 32-bit-wide device, supporting 8-bit, 16-bit, and 32-bit bus reads.

RAM

The card contains 1/2 megabyte (MP) of 16-bit-wide dynamic RAM. RAM is accessed by the 68000 and NuBus. When any device is accessed via NuBus, the 68000 is locked out from all access. RAM starts at location 000000, with the current 1/2 MB of RAM; the last RAM address is 07FFFF. When the 68000 accesses onboard RAM, no wait states are inserted.

To the NuBus interface, RAM appears as a full 32-bit wide device. RAM on the MCP card supports 8-bit, 16-bit, and 32-bit bus operations.

The operating system requires approximately 15 KB of memory on the MCP card.

Address map

Table 12-1 lists the various functions for the address spaces on the MCP card.

■ **Table 12-1** Address map

Address	Function
FF0000-FFFFFF	ROM (with two 256-Kbit ROMs, 64 KB)
F00000	Write - Place 68000 in RESET
E00000-EFFFFF	Test ROM (off card)
C0000A	Read - Set Interrupt IOP request
C00008	Read - Clear Interrupt IOP request
C00006	Read - Set Interrupt Host request
C00004	Read - Clear Interrupt Host request
C00002	Read - Clear Timer Interrupt
C00000	Read - Clear RESET
C00000	Write - NuBus Extension Register
A00000-BFFFFF	NuBus
800000-9FFFFF	I/O Interface Logic
400000-7FFFFF	I/O Interface Logic
080000-3FFFFF	Future RAM
000000-07FFFF	RAM (with 1/2 MB of RAM)

Timer

The MCP card provides an internal 6.5536 millisecond (ms) timer. Every 6.5536 ms, a level 1 interrupt occurs. This interrupt is cleared by reading location C00002.

△ **Important** If this interrupt is ignored for 3 ms, the next interrupt may not occur and a clock tick will be lost. △

Reset

The IOP can be placed in RESET by writing location `F00000` and placed out of reset by reading `C00000`. Any write to `FXXXXX` will place the 68000 in RESET, and any access to `CXXXXX` will take the 68000 out of RESET.

◆ *Note:* When NuBus resets, the 68000 comes out of RESET.

On power-on reset (NuBus reset), the first four accesses are fetched from the first four ROM locations (that is, the execution address and the stack pointer). Under "programmed" RESET, the address and stack pointer are fetched from RAM, starting at location 000000.

The start-up address vector in location 2 of the ROM must point to ROM address space (`F00000-FFFFF`).

Interrupts

Three interrupts are provided in the basic design: one for the timer, one for the NuBus interface, and one for the I/O interface. *Table 12-2* lists the interrupt priorities and provides a brief description of each:

■ **Table 12-2** Interrupt priorities

Interrupt	Level	Description
Timer	1	The I/O interface interrupt must remain asserted until the software resets this interrupt request.
NuBus	2	The IOP can interrupt the host by reading location C00006; this interrupt is cleared by the host reading location C00004.
I/O Interface	3	The IOP is interrupted by the host reading location C0000A; this interrupt is cleared by the 68000 reading location C00008.

NuBus interface

The NuBus interface provides for either master or slave operation. In master mode, the 68000 simply gains access to NuBus address space, and waits until the operation is complete. In slave mode, the 68000 is "removed" from the internal bus while the NuBus access is taking place.

Since the 68000 has an internal 16-bit bus, all bus cycles originating from the 68000 can be either 8-bit or 16-bit operations. This includes NuBus operations, where the 68000 is the NuBus master and both 8-bit and 16-bit operations are supported.

Special hardware has been included so that 32-bit access coming from NuBus will function correctly. The hardware performs two 16-bit bus operations on the 68000 bus whenever NuBus requests a 32-bit operation. As a result, the card supports 8-bit, 16-bit, and 32-bit NuBus transfers.

△ **Important** Two 68000 bus cycles are required for a 32-bit NuBus operation. Therefore, you should avoid using a 32-bit operation when only 16-bits are required, because of the increased amount of time required for the extra 68000 bus cycle. △

If the NuBus access cannot be completed, a bus error to the 68000 is reported.

NuBus address space

Access to the 32-bit NuBus address space is provided by a 12-bit address extension register. The most significant 12 bits of the NuBus address should be placed in this register before accessing the NuBus address space. This write-only register is located at location C000000.

In addition, the hardware uses A20 in the address field (not used for address calculation) to perform a read-modify-write cycle. Whenever a test-and-set instruction is executed, A20 must be set true. A20 should be set false for all other operations.

Acquiring the internal 68000 bus

An I/O front-end can insert itself in the BR/BG/BGACK daisy chain between the NuBus interface and the 68000. The I/O front-end can take over the 68000 bus and thus have full access to the resources on the card and NuBus. This gives the front-end the ability to talk to anything in the system that is on NuBus.

There is nothing in particular that the front-end must do to acquire NuBus; however, if the front-end does not provide its own extension register, the NuBus extension register must be loaded with the upper 12 address bits for any NuBus access. If the front-end provides its own dedicated NuBus extension register, there will not be any contention for the otherwise shared extension register.

◆ *Note:* The Programmable Array Logic (PAL) listing "DMA Example" in the next chapter is provided as an example for developers who may want to include DMA devices on the 68000 bus.

Design notes for NuBus

The following illustrations are provided to assist in your development efforts. For more details concerning NuBus, refer to *Designing Cards and Drivers.*

△ **Important** These examples do not pertain specifically to the MCP card, but are provided to assist you in designing your own NuBus interface. △

Figure 12-2 shows the function of various components, including arbitrating NuBus, generating the 68000 cycle when NuBus owns the local bus, decoding the slot, and so forth.

■ **Figure 12-2** MCP card functions

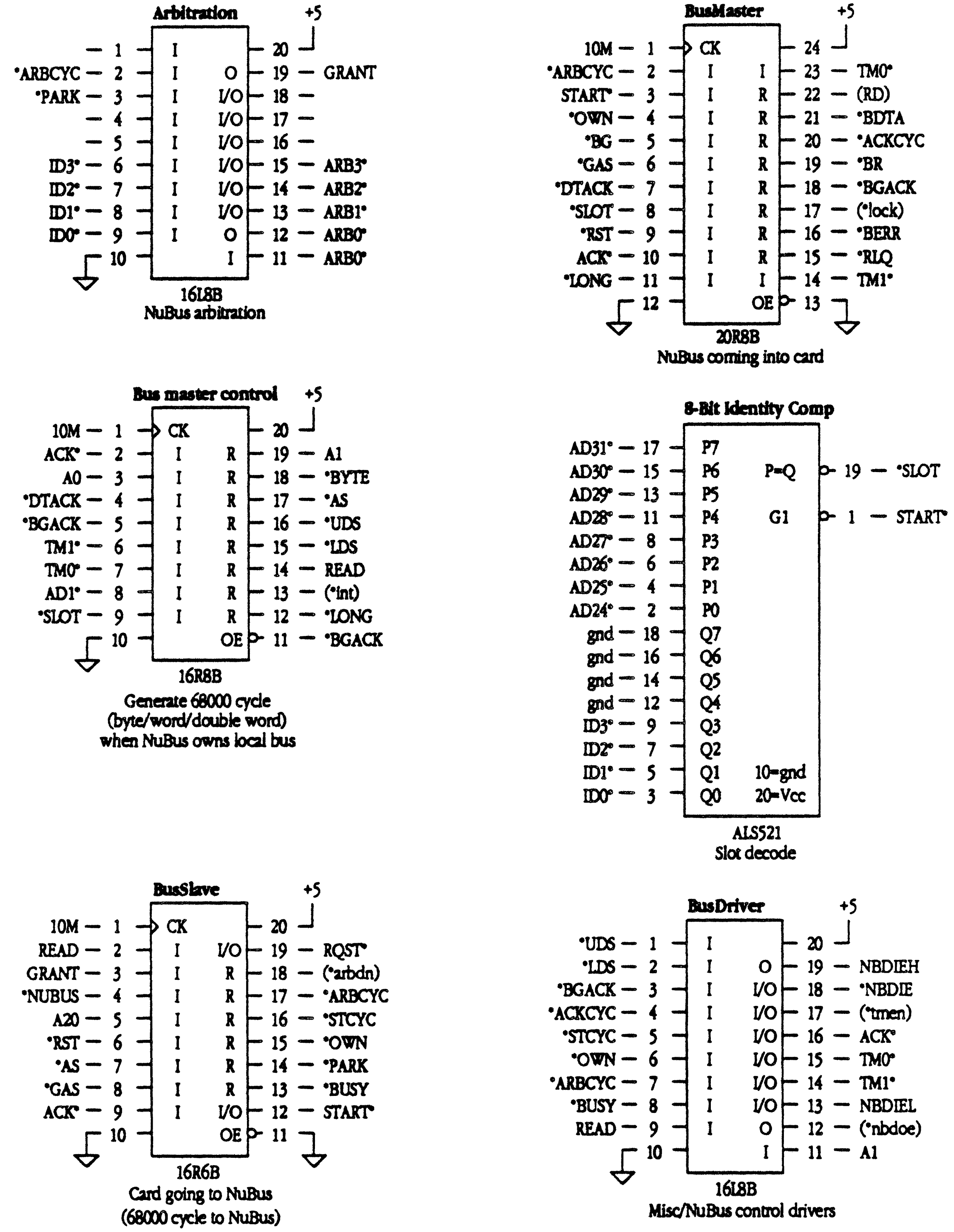


Figure 12-3 shows the generation of 20MHz and 10MHz clocks from the NuBus clock. Note that there is an equal delay from the NuBus clock for each of these cycles.

- **Figure 12-3** Generation of 20-MHz and 10 MHz clocks

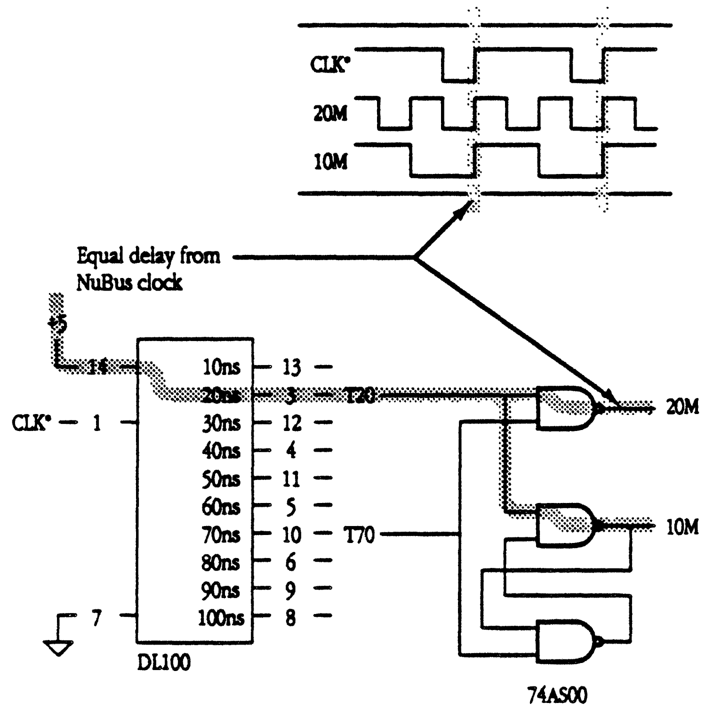
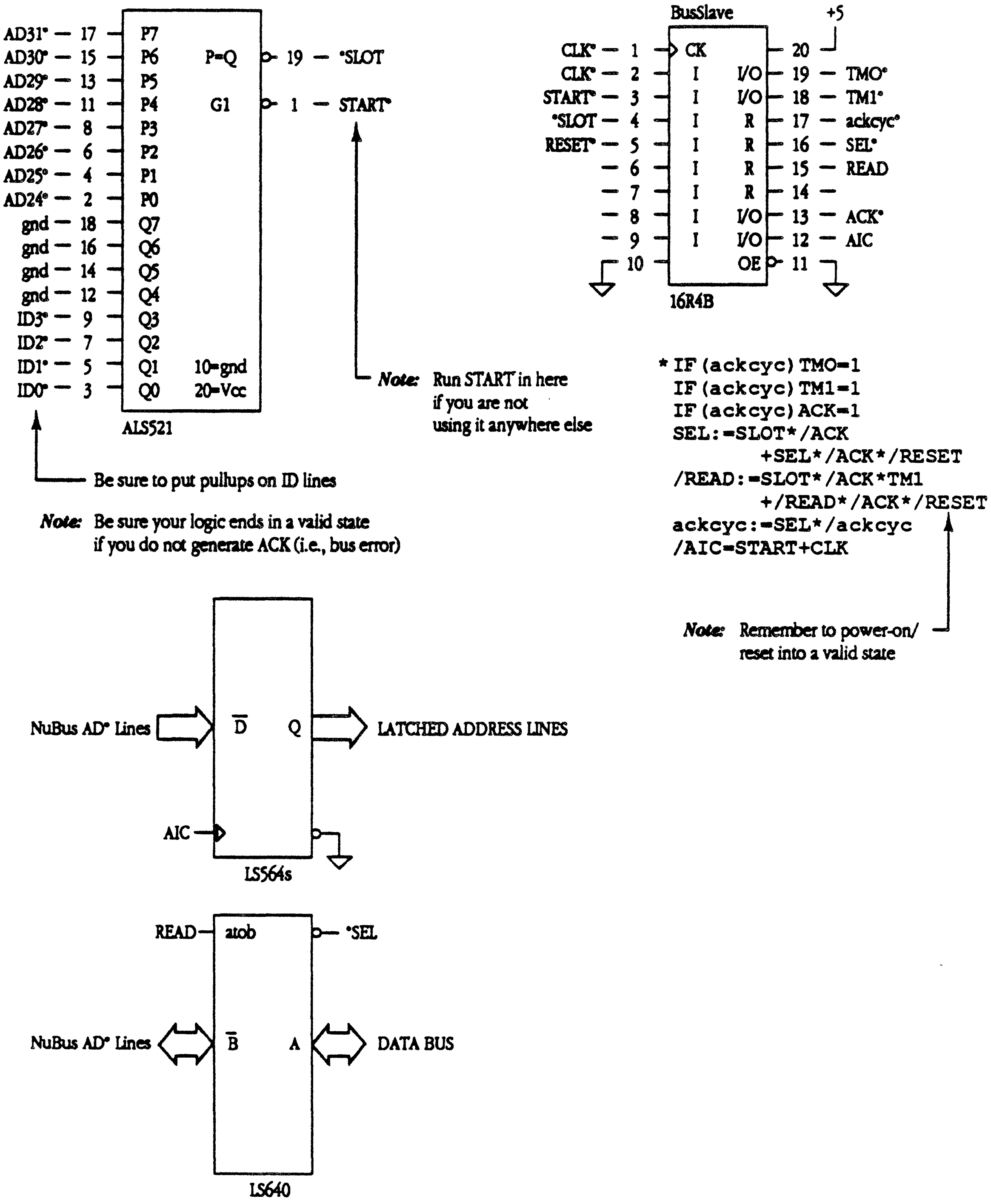


Figure 12-4 shows an example of a simple NuBus slave design, with explanatory notes.

- **Figure 12-4** A simple NuBus slave design



Note: Run START in here if you are not using it anywhere else

Be sure to put pullups on ID lines

Note: Be sure your logic ends in a valid state if you do not generate ACK (i.e., bus error)

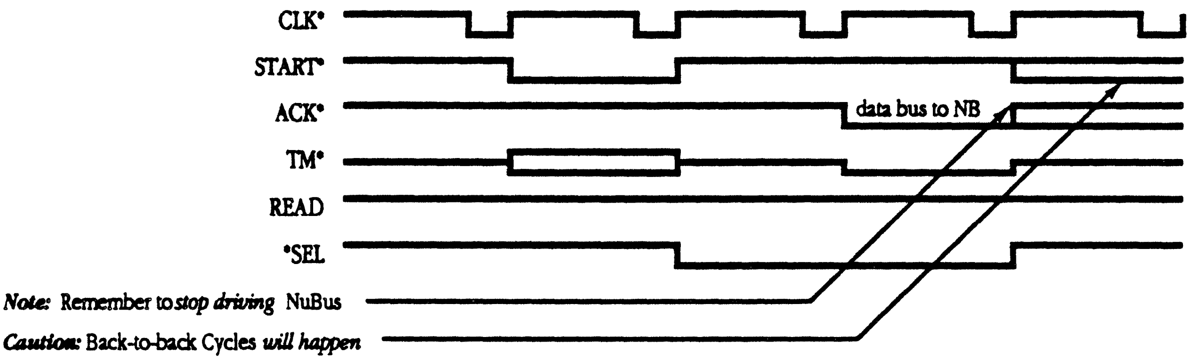
```
* IF (ackcyc) TMO=1
  IF (ackcyc) TM1=1
  IF (ackcyc) ACK=1
  SEL:=SLOT*/ACK
       +SEL*/ACK*/RESET
  /READ:=SLOT*/ACK*TM1
       +/READ*/ACK*/RESET
  ackcyc:=SEL*/ackcyc
  /AIC=START+CLK
```

Note: Remember to power-on/ reset into a valid state

Figure 12-5 shows the read and write timing cycles for the simple NuBus slave design shown in *Figure 12-4*.

- **Figure 12-5** Read and writing timing cycles

READ Cycle timing

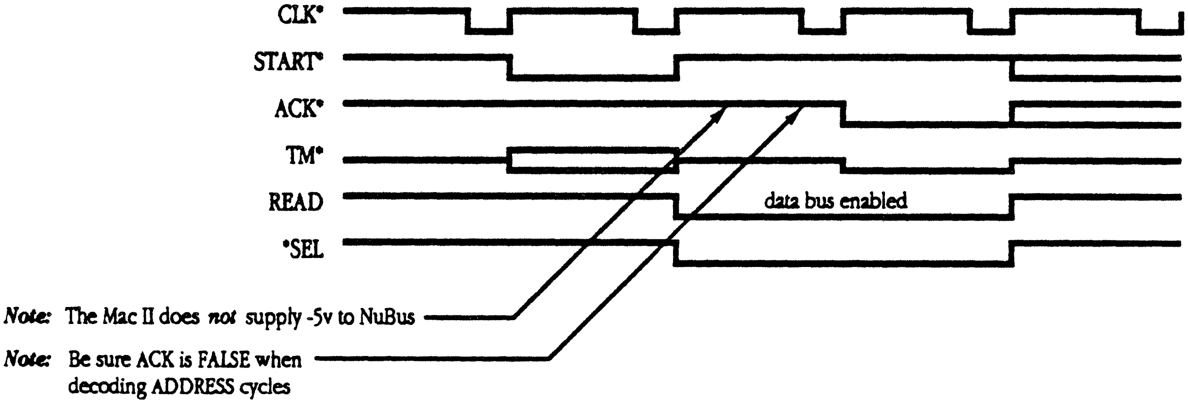


Note: Remember to ***stop driving*** NuBus

Caution: Back-to-back Cycles ***will happen***

READ: TM1=0
ACK: TM0=1
TM1=1

WRITE Cycle timing



Note: The Mac II does ***not*** supply -5v to NuBus

Note: Be sure ACK is FALSE when decoding ADDRESS cycles

WRITE: TM1=1
ACK: TM0=1
TM1=1

Chapter 13 **Listings for the MCP Card**

THIS CHAPTER provides listings for the PAL (Programmable Array Logic) equations and a parts list for the MCP card. ■

(The latest schematics for the MCP card are enclosed as separate pages at the back of this document. **<<Reviewers, should we delete this? Is this still true?>>**)

PAL listings

This section lists the equations for the PAL devices on the MCP card. These listings include equations for the following:

- Arbitration
- Bus driver
- Bus master
- Bus master control
- Bus slave
- Decode
- DMA example

◆ *Note:* This PAL listing is provided as an example for developers who may want to include DMA devices on the 68000 bus.

- Interrupts
- RAM (one row of RAM)
- RAM24 (two rows of RAM)

◆ *Note:* Use either RAM or RAM24, depending on your requirements for one or two rows of DRAM.

Each of these equations is more fully described in the next sections.

PAL equation: arbitration

The PAL equation for arbitration on the MCP card is listed below.

```
.IDENT     PAL16L8    Arb            {53E2}


      DATE:             7/7/87
      VERSION:          1A

.NAMES
 nc1    /AE1    /AE2     nc4     nc5    /ID3     /ID2     /ID1      /ID0     GND
/ARB0i  /ARB0o  /ARB1   /ARB2   /ARB3    arb0oe   arb1oe   arb2oe    GRANT   VCC

.EQUATIONS

.if[  AE1  *  AE2  *  ID3  ]
      ARB3  =  Vcc
        ;
   /arb2oe  =  /ID3  *  ARB3
        ;
.if[  AE1  *  AE2  *  ID2  *  arb2oe  ]
      ARB2  =  Vcc
        ;
   /arb1oe  =  /ID3  *  ARB3
        +  /ID2  *  ARB2
        ;
.if[  AE1  *  AE2  *  ID1  *  arb1oe  ]
      ARB1  =  Vcc
        ;
   /arb0oe  =  /ID3  *  ARB3
        +  /ID2  *  ARB2
        +  /ID1  *  ARB1
        ;
.if[  AE1  *  AE2  *  ID0  *  arb0oe  ]
     ARB0o  =  Vcc
        ;
    /GRANT  =  /ID3  *  ARB3
        +  /ID2  *  ARB2
        +  /ID1  *  ARB1
        +  /ID0  *  ARB0i
        ;
.END
```

PAL equation: bus driver

The PAL equation for the bus driver on the MCP card is listed below.

```
.IDENT    PAL16L8        BusDvr          {6F25}

     DATE:           5/18/88
     VERSION:        B

.NAMES
/UDS /LDS   /BGACK  /ACKCYC /STCYC /OWN   /ARBDN /BUSY    READ   GND
 A1  /NBDOE  NBDIEL /TM1     /TMO   /ACK   /tmen  /nbdie  NBDIEH VCC

.EQUATIONS
    tmen =  ACKCYC                           {enable TMx and ACK buffers}
       +  STCYC                            {delay th/ tmen drives lines
inactive}
       ;
.IF ( tmen )
     ACK =  ACKCYC
       +  STCYC *  BUSY                    {LOCK or UNLOCK}
       ;                                   {STCYC prevents glitch}
.IF ( tmen )
     TMO =  ACKCYC
       +  STCYC *  BUSY                    {LOCK or UNLOCK}
       +  STCYC * /ARBDN *  UDS * /LDS     {START - byte mode operation}
       +  STCYC * /ARBDN * /UDS *  LDS     {START - byte mode operation}
       ;
.IF ( tmen )
     TM1 =  ACKCYC
       +  STCYC * /ARBDN *  BUSY          {UNLOCK}
       +  STCYC * /ARBDN * /READ          {START - write operation}
       ;
  NBDOE  =  OWN    * /STCYC * /READ * BUSY {enable for master write}
       +  READ  *  ACKCYC                  {enable for slave read}
       ;
  nbdie  =  OWN   *  READ  * /STCYC          {we own nubus - master read}
       +  nbdie *  UDS                     {hold until DSs go away}
       +  nbdie *  LDS                     {hold until DSs go away}
       +  BGACK * /READ                    {bus owns us - slave write}
       ;
/NBDIEH  = /nbdie + /A1                      {high word}
       ;
/NBDIEL  = /nbdie +  A1                      {low word}
       ;
.NOTES
STCYC definitions:
          BUSY ARBDN   Function
           0     0     START
           0     1     IDLE
           1     0     UNLOCK
           1     1     LOCK
.END
```

PAL equation: bus master

The PAL equation for the bus master on the MCP card is listed below.

```
.IDENT     PAL20R8         BusMas                    {7A87}

     DATE:            9/19/88
     VERSION:         C

.NAMES
 10M   /NUBUS /START /OWN   /BG    /GAS    /DTACK /SLOT    /RST    /ACK /LONG GND
  en   /TM0    /RLQ    /BERR /lock /BGACK /BR     /ACKCYC /BDTA /rb   /TM1   VCC

.EQUATIONS

    rb :=  RST                                          {reset delayed for ICE}
  +  OWN * START * /ACK                          {busy for our mastership}
  +  OWN * rb     * /ACK                         {hold until ACK or null/attn}
  ;
  lock := /TM1 * START *  TM0 * ACK                    {LOCK from NuBus}
  + /RST * lock   * /TM0                         {hold until UNLOCK . . .}
  + /RST * lock   * /TM1
  + /RST * lock   * /START
  + /RST * lock   * /ACK
  ;
    BR :=  SLOT * /ACK    * /RST                        {START cycle to our slot}
  +  BR    * /BGACK * /RST                       {hold until BGACK}
  ;
 BGACK := /DTACK *   BR *   BG    * /GAS    * /OWN      {wait 'til everyone's done, own for
rmw}
  +  BGACK * /rb *   lock * /START               {if locked, hold until UNLOCK}
  +  BGACK * /rb *   lock * /TM0                 {if locked, hold until UNLOCK}
  +  BGACK * /rb *   lock * /TM1                 {if locked, hold until UNLOCK}
  +  BGACK * /rb *   lock * /ACK                 {if locked, hold until UNLOCK}
  +  BGACK * /rb * /lock * /ACK    * /ACKCYC    {if not locked, hold until any ACKCYC}
  +  BGACK * /rb * /lock *  START * /ACKCYC     {if not locked, hold until any ACKCYC}
  +  RST                                         {if 68K reset, we own bus}
  ;
ACKCYC :=  DTACK * BGACK * GAS * /LONG   * /ACKCYC  {when we get DTACK}
  ;
   RLQ :=  BR    *  NUBUS * GAS * /DTACK * /RLQ       {we want NuBus, NuBus wants us}
  +  RLQ   *  BERR                               {hold one clk past BERR(drvs halt)}
  ;
  BERR :=  BR    *  NUBUS * GAS * /DTACK * /RLQ       {we want NuBus, NuBus wants us}
  +  ACK   * /START * OWN * /TM0                 {TM0 & TM1 both asserted for OK op}
  +  ACK   * /START * OWN * /TM1                 {TM0 & TM1 both asserted for OK op}
  +  BERR * /RST    * GAS
  ;
  BDTA :=  OWN   * ACK * /START * TM0 * TM1           {DTACK pulse for master operation}
  +  BDTA * GAS * /RST
  ;

.END
```

PAL equation: bus master control

The PAL equation for bus master control on the MCP card is listed below.

```
.IDENT     PAL16R8       BMCtl        {6303}

     DATE:          9/30/87
     VERSION:       1A

 .NAMES
     10M  /ACK   A0  /DTACK /BGACK  /TM1  /TM0 /AD1  /SLOT  GND
     en   /LONG /int  READ  /LDS    /UDS  /AS  /byte  A1     VCC

.EQUATIONS

   byte  :=  SLOT *  TM0 * /ACK         {save byte/word mode for awhile}
       +  byte * /AS                 {and hold until AS}
       + /byte *  int * DTACK * AS   {used as 2nd internal state}
       ;
  /READ  :=  SLOT *  TM1 * /ACK        {set R/W from TM1}
       + /READ * /SLOT               {save until next access}
       ;
     AS  := /AS * /int *  BGACK         {start AS after BGACK-1st time, /int nth
time}
       +  AS * /int                  {and hold it...}
       +  AS *  int * /byte          {remove AS one state after DTACK}
       ;
    UDS  :=  READ * /byte * /int *  BGACK         {word read}
       +  READ *  byte * /int *  BGACK * /A0   {byte read}
       +  AS    * /byte * /int                 {word write}
       +  AS    *  byte * /int * /A0           {byte write}
       +  UDS   * /byte *  int                 {hold}
       ;
    LDS  :=  READ * /byte * /int *  BGACK
       +  READ *  byte * /int *  BGACK *  A0
       +  AS    * /byte * /int
       +  AS    *  byte * /int *  A0
       +  LDS   * /byte *  int
       ;
    int  := /int *  AS                       {internal state}
       +  int * /LONG *  BGACK * /SLOT    {if we keep 68K bus, hold int until SLOT}
       +  int * /LONG *  BGACK *  ACK     {...w/out ACK=addr cycle}
       +  int *  LONG * /byte             {first access of 32-bit operation}
       ;
    /A1  :=  SLOT * /AD1                     {set A1 at START cycle}
       + /SLOT * /A1   * /LONG            {hold until next SLOT or until...}
       + /SLOT * /A1   * /byte            {...last access of 32-bit access}
       ;
   LONG  :=  AS   * /int   * /byte * /A0 * /A1 {set for 32-bit NuBus operation}
       + /AS   *  LONG                    {hold until 2nd access starts}
       +  int  *  LONG                    {hold until 2nd access starts}
       ;
.END
```

PAL equation: bus slave

The PAL equation for the bus slave on the MCP card is listed below.

```
.IDENT     PAL16R6        BusSlv                        {93BF}
     DATE:          2/22/88
     VERSION:       A
.NAMES
 10M  READ   GRANT /NUBUS  A20   /RST   /AS      /GAS   /ACK   GND
 en   /START /BUSY  /PARK  /OWN  /STCYC /ARBCY  /arbdn /RQST   VCC

.EQUATIONS
 ARBCY :=  AS     * /GAS    *  NUBUS * /PARK                  {if don't own}
  +  AS    * /GAS   *  NUBUS *  RQST                    {if PARK going away next cycle rearb}
  +  AS    * /GAS   *  NUBUS *  A20                     {if RmW force rearb}
  +  GAS   * /PARK  *  ARBCY                            {hold while others arb}
  + /RST   *  PARK  *  ARBCY * /STCYC                   {hold until STCYC or UNLOCK}
  + /RST   *  GAS   *  NUBUS *  A20   *
               PARK  *  ARBCY * /BUSY *  STCYC          {hold during START for rmw}
  + /RST   *  ARBCY *  STCYC *  arbdn                            {hold during LOCK for rmw}
  ;
  PARK :=  AS     * /GAS    *  NUBUS * /PARK * /RQST           {if don't own}
  +  AS    * /GAS   *  NUBUS *  A20   * /RQST          {if RmW}
  +  ARBCY * /RQST                                      {if someone else arbing wait for RSQT fa
  +  PARK  * /RQST  * /RST                              {hold as long as no other RQSTs}
  +  PARK  *  ARBCY * /RST                              {hold as long as ARBCY}
  ;
   OWN :=  ARBCY *  GRANT *  arbdn * /OWN * /BUSY              {always take after arb}
  +  ARBCY *  GRANT *  arbdn * /OWN *  ACK              {always take after ack}
  +  AS    * /GAS    *  NUBUS * /A20 * /RQST  *  PARK   {norm parked}
  +  ARBCY *  OWN    * /RST                             {hold if we buslock, until UNLOCK}
  + /ARBCY *  OWN    * /ACK    * /RST * /arbdn          {if not rmw, OWN goes away with ACK}
  ;
 STCYC :=  ARBCY *  GRANT *  arbdn * /OWN * /BUSY              {always take after arb}
  +  ARBCY *  GRANT *  arbdn * /OWN *  ACK              {always take after ack}
  + /GAS * AS * /A20 * NUBUS * /OWN * /RQST  *  PARK    {norm parked}
  +  STCYC *  BUSY  *  arbdn                            {take after LOCK (read of RmW)}
  +  GAS   *  A20   * /READ  *  OWN * /STCYC * /BUSY    {write of RmW}
  +  ARBCY *  ACK   * /READ  *  OWN * /START            {UNLOCK of rmw}
  +  STCYC *  BUSY  * /arbdn                            {after UNLOCK do IDLE}
  + /STCYC *  OWN   * /GAS   * /AS                      {UNLOCK if OWN w/out GAS,rmw failed}
  ;
  BUSY := /BUSY  * /ACK   *  START                            {start on START cycle}
  +  BUSY  * /ACK   * /RST                              {hold 'til ACK}
  +  ARBCY *  ACK   * /READ  *  OWN * /START            {UNLOCK of rmw}
  +  arbdn *  GRANT * /OWN   *  GAS *  NUBUS * A20      {LOCK if rmw still there}
  +  arbdn *  GRANT * /OWN   *  ACK * /GAS              {UNLOCK if GAS gone}
  +  arbdn *  GRANT * /OWN   *  ACK * /NUBUS            {UNLOCK if NUBUS gone}
  + /STCYC *  OWN   * /GAS   * /AS                      {UNLOCK if OWN w/out GAS,rmw failed}
  ;
 arbdn := /arbdn *  ARBCY *  PARK  * /OWN * /START
  +  arbdn *  GRANT *  BUSY  * /ACK                     {hold if busy, release on ACK}
  +  arbdn *  GRANT * /OWN   *  GAS *  NUBUS * A20     {LOCK if rmw still there}
  + /arbdn *  STCYC *  BUSY  *  GAS *  NUBUS * A20     {goto IDLE from UNLOCK after rmw}
  + /arbdn *  STCYC *  BUSY  * /NUBUS                  {goto IDLE from UNLOCK if no nubus r
  + /arbdn *  STCYC *  BUSY  * /GAS                         {else do START}
  ;
.IF ( OWN )         START  =  STCYC *  BUSY                 {Drive START while we own bus}
                       +  STCYC * /arbdn                {assert during STCYC except IDLE cyc
  ;
.IF ( ARBCY * PARK { * /STCYC } )    RQST  =  Vcc            {hold RQST until start cycle}
  ;
.END
```

PAL equation: decode

The PAL equation for decoding on the MCP card is listed below.

```
.IDENT     PAL20R4        Decode                       {80F7}

     DATE:           1/6/88
     VERSION:        2A

.NAMES
 10M  A23      A22    A21     A20  /AS      READ    /RESET  /GAS    FC1  FC0  GND
 en  /RFCYC   /CR    /NUBUS   WAL  /LRST   /setup   /DDTA   /VPA   /ROM  nc23 VCC

.EQUATION

  ROM   =  A23 *  A22 *  A21 *  A20 *  AS *  READ * /VPA     {ROM space decode}
       + /A23 * /A22 *                 AS *  READ *  setup   {ROM at setup}
       ;
   CR   =  A23 *  A22 * /A21 * /A20 *  AS *  READ     {ctl reg read - CXXXXX}
       ;
 /WAL  := /A23 + /A22 +  A21 +  A20 + /AS +  READ     {write addr latch}
       ;
setup  :=  RESET
       +   setup * /AS
       +   setup * /A23                               {as long as in low 8mb}
       ;
 LRST  :=  A23   *  A22 *  A21 * A20 *  AS   * /READ  {set RST w/ write to F00000}
       +   LRST * /CR   * /RESET                      {clear reset w/ CR read}
       ;
 DDTA  :=  GAS *  A23 *  A22 *  A21   * /VPA          {E00000-FFFFFF - ROMs}
       +   GAS * /A23 * /A22 *  setup                 {000000-3FFFFF - ROM w/ setup}
       +   AS   *  A23 *  A22 * /A21   * /A20         {C00000-CFFFFF - ctl reg}
       +   GAS * DDTA *  A23                          {good hold, not RAM}
       +   AS   * /A23 * /A22 * /setup * /RFCYC       {000000-3FFFFF - RAM}
       +   GAS * DDTA * /RFCYC                        {good hold, RAM, for rmw}
       ;
 .IF    ( GAS *  FC0 * FC1 {* /NBACK} )
  VPA   =  AS
       ;
NUBUS   =  A23 * /A22 * A21                           {NuBus = A00000-BFFFFF}
       ;

.END
```

PAL equation: DMA example

An example of a PAL equation for providing DMA on the MCP card is listed below.

```
       DATE:            2/22/88
       VERSION:         1.1

.NAMES

/BG    /NBACK /XBACK /NBR  SRE /XBR    /STCYC   /XDE     A22  GND
/RST   /XBG    /NBG    /nbn /BR   /BGACK /STCY0   /STCY1   nc19 VCC

.EQUATIONS
   STCY0 =  STCYC *  NBACK * /SRE        {internal, normal AB}
       +  STCYC *  SRE    * /A22       {int/ext, seperate AB}
       ;
   STCY1 =  STCYC * /NBACK * /SRE        {external, normal AB}
       +  STCYC *  SRE    *  A22       {int/ext, seperate DE}
       ;
.IF ( XDE )
      XBG = BG   * /nbn * /NBACK * /NBG
       + XBG *  BG * XDE
       ;
      NBG = nbn *  BG   * /XBACK
       + nbn *  BG   * /XDE              {*if XDE dissabled}
       + NBG *  BG
       ;
      nbn = NBR * /BG
       + nbn * /NBG * /RST
       ;
       BR = NBR
       + XBR *  XDE
       ;
   BGACK = NBACK
       + XBACK * XDE
       ;
.NOTES
This PAL can be placed between the MCP logic and the 68000 and adds external DMA
arbitration logic:
       BR,  BG,  & BGACK go to 68000
       NBR, NBR, & NBACK go to MCP logic
       XBR, XBG, & XBACK go to external logic
       STCY0 & STCY1 are used if a second NuBus extension register is added.

.END
```

PAL equation: interrupt

The PAL equation for interrupts on the MCP card is listed below.

```
.IDENT     PAL16R4         Int                         {5DA6}


     DATE:             7/15/87
     VERSION:          1A

.NAMES
      10M /CR     A3     A2      A1     /RST    /IOIR     TMR  MUX    GND
     /EN   /IPL0 /IPL1 /TMRIR /IOPIR /HSTIR /tmrdly /RA0 /NMR    VCC

.EQUATIONS

    IPL0  =  IOIR                                {timer = level 1}
       +   TMRIR * /IOPIR                    {NuBus = level 2}
       ;                                     {I/O   = level 3}
    IPL1  =  IOIR
       +   IOPIR
       ;
.IF ( HSTIR )
    NMR   =  Vcc
       ;
  tmrdly :=  TMR
       ;
     RA0  =  MUX * A2
       + /MUX * A1
       ;
   TMRIR :=  TMR * /tmrdly * /RST                   {Addr=2 clr, set by timer}
       + /CR   *   TMRIR   * /RST
       +  A3   *   TMRIR   * /RST
       +  A2   *   TMRIR   * /RST
       + /A1   *   TMRIR   * /RST
       ;
   HSTIR := /A3   *  A2      *  A1   *  CR * /RST     {Addr=4 clr, 6 set}
       + /CR   *   HSTIR   * /RST
       +  A3   *   HSTIR   * /RST
       + /A2   *   HSTIR   * /RST
       ;
   IOPIR :=  A3   * /A2      *  A1   *  CR * /RST     {Addr=8 clr, A set}
       + /CR   *   IOPIR   * /RST
       + /A3   *   IOPIR   * /RST
       +  A2   *   IOPIR   * /RST
       ;
.END
```

PAL equation: RAM

The PAL equation for RAM on the MCP card is listed below.

```
.IDENT     PAL16R6        RAM                  {99A9}

     DATE:           9/20/88
     VERSION:        D

.NAMES
     20M    10M    A22   /AS    /UDS     /LDS     READ    13us   /SETUP   GND
     en     A23   /rfd   /GAS   /rfcyc   /CASH   /CASL   /MUX    /RAS     VCC

.EQUATIONS

     GAS := /10M * AS   * /READ                  {first time write, w/ AS}
       + /10M * UDS                     {first time read,  w/ DS}
       + /10M * LDS                     {first time read,  w/ DS}
       +  GAS * UDS                     {hold with DS, for RmW to 8-F, 2 GAS}
       +  GAS * LDS                     {hold with DS, for RmW to 8-F, 2 GAS}
       +  GAS * AS   * /A23             {hold with AS, for RAM to 0-7, 1 GAS}
{C}        +  GAS * AS   * /READ                 {hold with AS, for for write}
       + /10M * GAS               {aways hold on this edge}
       ;
     RAS  = /rfcyc *  AS   * /A23   * /A22   * /CASL * /CASH * /SETUP
       +  MUX
       ;
     MUX := /rfcyc * /10M  * /CASL * /CASH * AS * /A23 * /A22 * /SETUP * /MUX
{D}        + /rfcyc *  MUX  * /CASL * /CASH * GAS  {GAS added rev D}
       + /rfcyc *  MUX  * /10M
       +  rfcyc * /rfd  *  CASL                 {CAS=MUX during refresh}
       +  rfcyc *  10M  *  MUX
       ;
    CASL := /rfcyc *  LDS  *  MUX  *  READ          {CAS on read early}
       + /rfcyc *  LDS  *  MUX  * /10M          {CAS on write late}
       + /rfcyc *  CASL *  MUX
       + /rfcyc *  CASL * /10M
{D}        + /rfcyc *  CASL *  LDS
       +  rfcyc *  rfd  * /10M                  {refresh}
       +  rfcyc * /rfd  *  10M  *  CASL
       ;
    CASH := /rfcyc *  UDS  *  MUX  *  READ          {CAS on read early}
       + /rfcyc *  UDS  *  MUX  * /10M          {CAS on write late}
       + /rfcyc *  CASH *  MUX
       + /rfcyc *  CASH * /10M
{D}        + /rfcyc *  CASH *  UDS
       +  rfcyc *  rfd  * /10M                  {refresh}
       +  rfcyc * /rfd  *  10M  * CASH
       ;     rfd :=  13us  * /rfcyc                              {rfcyc for RMW only}
       +  rfd   * /rfcyc
       +  rfd   *  10M
       + /rfcyc *  CASH * /UDS  * /LDS *  AS *  10M   {RMW force refresh cycle}
```

```
        + /rfcyc *  CASL * /UDS  * /LDS *  AS *  10M   {in b/twn r&w to add delay}
        ;
    rfcyc := /13us  *  rfd  * /10M  * /AS  * /GAS        {to meet su in both dirs}
        + /13us  *  rfd  * /10M  *  AS  *  A23          {ok if not RAM access}
        + /13us  *  rfd  * /10M  *  AS  *  A22          {ok if not RAM access}
        +  rfcyc *  10M
        +  rfcyc *  rfd
        +  rfcyc *  MUX
        + /rfcyc *  CASH * /UDS  * /LDS *  AS *  10M   {RMW force refresh cycle}
        + /rfcyc *  CASL * /UDS  * /LDS *  AS *  10M   {in b/twn r&w to add delay}
        ;
.END
```

PAL equation: RAM24

The PAL equation for RAM24 on the MCP card is listed below.

```
.IDENT       PAL20R6           RAM24           {D5DA}
       DATE:              9/20/88
       VERSION:          D

.NAMES
       20M   10M    nc3    /AS     /UDS   /LDS    READ    13us   /SETUP   A23    A22    GND
       en    A19   /RASH   /rfcyc /GAS   /rfd   /CASH   /CASL   /MUX    /RASL   nc23   VCC

.EQUATIONS
       GAS := /10M * /GAS * AS * /READ    {first time write, w/ AS}
        + /10M * /GAS * UDS              {first time read,  w/ DS}
        + /10M * /GAS * LDS              {first time read,  w/ DS}
        +  10M *  GAS * UDS              {hold with DS, for RmW to 8-F, 2 GAS}
        +  10M *  GAS * LDS              {hold with DS, for RmW to 8-F, 2 GAS}
        +  10M *  GAS * AS * /A23        {hold with AS, for RAM to 0-7, 1 GAS}
{C}          +  10M *  GAS * AS * /READ    {hold with AS, for for write}
        + /10M *  GAS                    {aways hold on this edge}
        ;
       RASH = /rfcyc *  AS   * /A23  * /A22  *  A19  * /CASL * /CASH * /SETUP
        + /rfcyc *  MUX  *  A19
        +  rfcyc *  MUX
        ;
       RASL = /rfcyc *  AS   * /A23  * /A22  * /A19  * /CASL * /CASH * /SETUP
        + /rfcyc *  MUX  * /A19
        +  rfcyc *  MUX
        ;
       MUX := /rfcyc * /10M  * /CASL * /CASH * AS * /A23  * /A22 * /SETUP * /MUX
{D}          + /rfcyc *  MUX  * /CASL * /CASH * GAS  {GAS added rev D}
        + /rfcyc *  MUX  * /10M
        +  rfcyc * /rfd  *  CASL                 {CAS=MUX during refresh}
        +  rfcyc *  10M  *  MUX
        ;
     CASL := /rfcyc *  LDS  *  MUX  *  READ         {CAS on read early}
        + /rfcyc *  LDS  *  MUX  * /10M         {CAS on write late}
        + /rfcyc *  CASL *  MUX
        + /rfcyc *  CASL * /10M
{D}          + /rfcyc *  CASL *  LDS
        +  rfcyc *  rfd  * /10M                {refresh}
        +  rfcyc * /rfd  *  10M  *  CASL
        ;
     CASH := /rfcyc *  UDS  *  MUX  *  READ         {CAS on read early}
        + /rfcyc *  UDS  *  MUX  * /10M         {CAS on write late}
        + /rfcyc *  CASH *  MUX
        + /rfcyc *  CASH * /10M
{D}          + /rfcyc *  CASH *  UDS
        +  rfcyc *  rfd  * /10M                {refresh}
        +  rfcyc * /rfd  *  10M  * CASH
        ;
     rfd :=  13us   * /rfcyc                                {rfcyc for RMW only}
       +  rfd    * /rfcyc
       +  rfd    *  10M
       + /rfcyc *  CASH * /UDS   * /LDS *  AS *  10M   {RMW force refresh cycle}
       + /rfcyc *  CASL * /UDS   * /LDS *  AS *  10M   {in b/twn r&w to add delay}
       ;
   rfcyc := /13us   *  rfd  * /10M   * /AS   * /GAS        {to meet su in both dirs}
       + /13us   *  rfd  * /10M  *  AS   *  A23        {ok if not RAM access}
       + /13us   *  rfd  * /10M  *  AS   *  A22        {ok if not RAM access}
       +  rfcyc *  10M
       +  rfcyc *  rfd
       +  rfcyc *  MUX
       + /rfcyc *  CASH * /UDS   * /LDS *  AS *  10M   {RMW force refresh cycle}
       + /rfcyc *  CASL * /UDS   * /LDS *  AS *  10M   {in b/twn r&w to add delay}
       ;
.END
```

Parts for the MCP card

Table 13-2 lists the parts required for the MCP smart card, along with the quantity required and a brief description of each part.

■ **Table 13-1** Parts list for the MCP card

Quantity	Name	Description
1	Capacitor	Electrolytic,10 UF 16V
30	Capacitor	Ceramic, Axial .01UF 20% 50V
1	Connector	Header, Right Angle, Euro DIN 3-Row 96-Pin
1	Delay Line	24P, 20 TAP Delays 100NS
4	IC	44C256 (DIP Package)
1	IC	68000, CPU, 12.5 MHz
1	IC	74ALS02
1	IC	74ALS09 Quad 2-Input
1	IC	74ALS521, 8-bit Identity Comp
1	IC	74ALS563, Octal D-Type
1	IC	74ALS564, Octal D-Type
6	IC	74ALS651
1	IC	74ALS880, Dual 4-Bit D-Type
1	IC	74AF00, Quad 2-Input Nand Gate
2	IC	74ALS258
2	IC	74LS590, 8-bit Binary Counter
2	IC	EPROM, 32K x 8, 250NS
5	Resistor	1KB OHM 1/4W 5%
3	Resistor Pak	47 OHM, 10 POS
1	Resistor Pak	Network 9 x 3.3K OHM 5%
10	Socket	IC, 20-Pin
2	Socket	IC, 24-Pin
1	Socket	IC, 64-Pin
2	Socket	PLCC, 28-Pin
1	Switch	KeyType
1	PAL	16L8B (Arbitration)
1	PAL	16L8B (Bus driver)
1	PAL	16R4A (Interrupt)
1	PAL	20R4B (Decode)
1	PAL	16R6B (RAM)
1	PAL	16R6B (Bus slave)
1	PAL	16R8B (Bus master control)
1	PAL	20R8B (Bus master)

Chapter 14 **Diagnostics for the MCP Card**

This chapter provides an overview of the diagnostics available for MCP 1.1.

MCP card declaration ROM

The MCP card declaration ROM is divided into several parts:

- the on-card power-up tests
- the primary initialization code (run by the Macintosh II system at boot time)
- the application-specific resources
- the application-specific drivers

You can use the hooks available in the power-up and initialization sections of the ROM to insert your own application-specific code into the test sequence.

Power-up diagnostics

When power reaches the card (or upon a software reset), the on-board 68000 power-up tests automatically begin execution. Before execution, all interrupts are disabled by the MCP hardware. The tests

- verify the 68000 data and address lines
- check CRC of the Declaration ROM
- check critical functionality of on-board RAM
- clear RAM memory from $180 to $7FFFE (that is, the last half-megabyte)

The tests are implemented so that, if a test crashes as a result of hardware problems, the failure is still reported in low memory and to the Slot Manager.

If these tests pass, the 68000 exception vector table is initialized and the on-board RAM size is stored in low memory (currently $11C-F). The timer interrupt (level 1) vector points to a routine that increments a 32-bit counter at location $118 every 6.5536 milliseconds. The Non-Maskable Interrupts (NMI), wired to the button on the prototype MCP card, are vectored to a routine that restarts the power-up code by simulating a reset. You can change these default interrupts using the file `ApplPowerOn.a` (described in the next section).

Next, the level 1-7 auto vector interrupts are enabled to the routines defined in the file `ApplPowerOn.a`. Then the code executes a test reserved for the application you develop. Currently, this is a stub function named `VendorPowerUp` in the file `ApplPowerOn.a`. If you insert any code here, it must signal success or failure by returning a bit flag into the test status location.

All tests have an associated bit flag. These flags are kept in a word at location $102. At the start of the power-up code, all bits in the flag word are set. To indicate success, the bit flag associated with that test is cleared. In the case of the developer test, bit 4 (the $0010 bit) is the associated bit. Any code you insert here should not take more than 600 milliseconds, because a software reset causes the 68020 to execute an abbreviated memory test, thereby shortening the time between reset and 68020 primary initialization.

When the power-up code is finished, a wait flag is cleared at word location $100. Next, the 68000 executes a `STOP` instruction with interrupts enabled to wait for the primary initialization tests.

68020/030 primary initialization tests

The primary initialization code is run by the Macintosh II operating system at the time of system initialization. The code is read off of the declaration ROM and executed on the card across NuBus. Any application code that you add must take this into account.

The primary initialization code tests NuBus and the interrupt system for the MCP card. The code begins by getting the results of the power-up tests. If these have passed, the primary initialization code then tests

- 32 bit data line test across NuBus to the card's RAM
- the ability of the Macintosh II to reset the 68000
- the timer interrupt
- the ability of the Macintosh II to interrupt the on-board processor via a NuBus interrupt.

After this, any routines you supply are executed. Currently, there is a stub routine in the file `ApplPrimaryInit.a`, but any application-specific initialization should either be done here or during the device driver Open function.

△ **Important** The primary initialization code for the discrete PAL version of the MCP card must currently reside at the very end of the declaration ROM before the format/header block. In the current version, some of the test subroutines reside at specific addresses and must not be moved. △

If you want to put your own code in the `VendorInit` routine, you must be sure to indicate whether the routine has passed or failed. The primary initialization code expects the D1 register to return $00 if the test has passed and -1 if it has failed. The A2, A3, and D0 registers must be preserved.

Data area

The power-up and primary initialization code have a data area in the on-board RAM that starts at location $100 and extends to $150.

△ **Important** If any application uses this area of memory (such as A/ROSE), these values are destroyed. △

Locations $100 through $14F are reserved for existing code; locations $150 through $180 are reserved for developers. Certain locations are reserved for use by the application-specific code on the ROM. *Table 14-1* identifies the data areas and briefly describes each.

▲ **Warning** The ROMs provided on the MCP card will overwrite these locations. However, this will not occur when you build your own ROMs, since the source code provided on the distribution disk that you will use to build your own ROM has fixed this problem. ▲

■ **Table 14-1** Data area

Location	Description
$102 - $103	Tests status bit flags. This word holds the bit flags used to track the power-up code. A-32 in this location means that all power-on tests have passed. The first five error codes listed in Table 14-4 are found in this location.
$108 - $109	Signals a soft reset. This word is set to $FFFF when the primary initialization code has finished.
$10E - $111	Contains the CRC checksum calculated by the power-on ROM.
$118 - $11B	Used as a timer tick counter and is incremented every 6.5536 milliseconds.
$11C - $11F	Contains the amount of RAM on the card in bytes.
$134 - $137	Used by vendor to pass back information about power-on test other than `PASS` or `FAIL`.
$138 - $13B	Used by vendor to pass back more information about primary-initialization test other than `PASS` or `FAIL`.
$13C - $13F	Stores the 68000 Program Counter here after any hardware exception.
$140 - $143	Stores the address that the 68000 was trying to access, when a hardware exception occurs.
$150 - $180	Reserved for developers.

Error codes

Table 14-2 lists the codes returned to the Slot Manager by the primary initialization code. At the end of the primary initialization code, if any test has failed, the bit flags are returned as a negative number. The Slot Manager stores this number in an array.

To find the error code, use the Macintosh toolbox call `sReadInfo` (refer to the chapter on the Slot Manager in *Inside Macintosh, Volume 5*); the value of the error code for the MCP_Diagnostic is returned in the `sInitStatusV` field.

◆ *Note:* These error codes are applicable only for Revision D ROMs.

■ **Table 14-2** Error codes

Error code	Description
0	All tests passed
-1	Data Line test failed
-2	ROM test pattern not found
-6	CRC test failed
-14	RAM test failed
-16	Vendor power-up test failed
-20	power-on tests did not complete
-32	NuBus data line test failed
-64	IOP interrupt test failed (Level 2)
-320	Host reset test failed
-384	Timer interrupt test failed (Level 1)
-512	Vendor initialization test failed

Note: Since interrupts cannot be enabled during primary initialization, the NMRQ (card-to-Macintosh interrupt) cannot be tested during primary init.

Appendix A **Files on the MCP Distribution Disks**

For your information, this appendix provides a list of folders and files on the MCP distribution disks. Be sure to check the actual distribution disks for accurate, up-to-the-minute listings of files and folders.

There are two distribution disks provided for Version 1.1 of the Macintosh Coprocessor Platform:

- *A/ROSE 1*
- *A/ROSE 2*

Files on *A/ROSE 1*

Table A-1 lists the folders and files found on the distribution disk named *A/ROSE 1* and provides a brief description of each. The folder name provides a complete description of the pathname to the file.

▲ **Caution** Diagnostic code is still under development. ▲

■ **Table A-1** Files on *A/ROSE 1*

File name	**Description**
FOLDER: A/ROSE 1:A/ROSE:	
`:AST_ICP:`	Folder containing files and folders tailored to the AST-ICP card
`:Examples:`	Folder containing example files and folders
`:includes:`	Folder containing includes files
`:MCP:`	Folder containing files and folders tailored to the MCP card

■ **Table A-1** Files on *A/ROSE 1* *continued*

File name	Description
`L3MMSVP.c`	The source of C routines comprising part of the hardware diagnostic task MMSVP
`L3MMSVPClient.c`	The source of the task, `L3MMSVPClient`, designed to control the hardware diagnostic task MMSVP
`Makefile`	MakeFile makes all examples and test tasks found within the :A/ROSE:Examples: folder
`:MCP:`	a folder containing example files for the MCP card
`name_tester.c`	The source of the test task designed to test the Name Manager
`osmain.c`	The source of the initialization routine that makes calls to initialize A/ROSE, initializes any hardware that needs initialization before any tasks start executing, specifies tasks to be initially started when A/ROSE starts executing, and starts A/ROSE executing
`ossccint.a`	The source of a set of example interrupt handler routines. These interrupt handler routines handle interrupts from an SCC chip. Please see the `pr_manager.c` example
`printf.c`	The source of the subroutine that performs the text formatting functions of the standard C `printf` routine (this subroutine also looks for a print manager and requests that the Print Manager print the text that it has formatted)
`pr_manager.c`	The source of a task that controls an SCC chip running in an asynchronous mode. This task receives a message from another task requesting that text be printed and sends a reply to the requesting task when the text is printed. This task uses interrupt handler routines found in `ossccint.a`
`timeIt.c`	The source of a test program that measures the amount of time it takes for messages to be sent between itself and a echo manager. The Echo Manager may be on the same card as this test program, on a different card, or in the Macintosh II (slot 0)
`timer_tester.c`	The source of a test program designed to test the Timer Manager
`trace_manager.c`	The source of a software diagnostic program that can be used to trace messages sent between tasks

■ **Table A-1** Files on *A/ROSE 1* *continued*

File name	Description
FOLDER: A/ROSE 1:A/ROSE:Examples:AST_ICP:	
FOLDER: A/ROSE 1:A/ROSE:Examples:Binaries:	
`Download`	An MPW tool designed to download a module to the card. This module contains the A/ROSE operating system and any tasks or managers that are to be downloaded with the A/ROSE operating system.
`dumpcard`	An MPW tool designed to dump information about a card that is or was running A/ROSE. This tool is meant to assist in trouble-shooting problems. It dumps the global common area of A/ROSE, task control block information for the tasks running under A/ROSE, and other information
`echo.c.o`	The object file of the `echo.c` routine compiled to run on the card
`L3MMSVP.a.o`	The object file of assembler routines comprising part of the hardware diagnostic task MMSVP

Table A-1 Files on *A/ROSE 1* *continued*

File name	Description
L3MMSVP.c.o	The object file of C routines comprising part of the hardware diagnostic task MMSVP
L3MMSVPClient.c.o	The object file of the task, L3MMSVPClient, designed to control the hardware diagnostic task MMSVP
map	The map produced by Link during the building of the start module for the card
name_tester.c.o	The object file of the name_tester.c routine compiled to run on the card
osmain.c.o	The object file of the osmain.c routine compiled to run on the card
ossccint.a.o	The object file of the ossccint.a routine compiled to run on the card
printf.c.o	The object file of the printf.c routine compiled to run on the card
pr_manager.c.o	The object file of the pr_manager.c routine compiled to run on the card
start	The module produced by Link during the use of the MakeFile in folder :A/ROSE:Examples: for building an example to be downloaded to the card; this module contains the initialization routine osmain.c, the version of MCP operating system designed to run on the card, and numerous test tasks
timeIt.c.o	The object file of the timeIt.c routine compiled to run on the MCP card
timer_tester.c.o	The object file of the timer_tester.c routine compiled to run on the card
trace_manager.c.o	The object file of the trace_manager.c routine compiled to run on the card
xref	The cross reference produced by Link during the building of the start module for the card

■ **Table A-1** Files on *A/ROSE 1* *continued*

File name	**Description**
FOLDER: A/ROSE 1:A/ROSE:includes	
`clister.h`	Include file that defines dummy macros for a program clister
`diags.a`	Include file that contains constants used by the hardware diagnostic programs `MMSVP` and `MMSVPClient`
`diags.h`	Include file that contains constants used by the hardware diagnostic programs `MMSVP` and `MMSVPClient`
`Download.h`	Include file that contains constants and definitions used when calling the Download and Findcard subroutines
`iccmDefs.a`	Include file provided for debugging purposes only that contains constants and definitions used by ICCM
`iccmDefs.h`	Include file provided for debugging purposes only that contains constants and definitions used by ICCM
`managers.a`	Include file that contains constants and definitions used when sending message requests to the A/ROSE managers (such as ICCM, Name Manager, and others)
`managers.h`	Include file that contains constants and definitions used when sending message requests to the A/ROSE managers(such as ICCM, Name Manager, and others)
`mrdos.a`	Include file that contains constants and definitions used by the A/ROSE operating system, as well as the definition of the global common area
`mrdos.h`	Include file that contains constants and definitions used by the A/ROSE operating system, as well as the definition of the global common area
`os.a`	Include file that contains constants and definitions and macros used when invoking A/ROSE primitives (those functions within A/ROSE invoked by instruction traps and include `GetMsg`, `GetMem`, `Send`, `Reschedule`, and others)
`os.h`	Include file that contains constants and definitions and external routine declarations used when calling A/ROSE primitives and utility routines (primitives are those functions within A/ROSE invoked by instruction traps and include `GetMsg`, `GetMem`, `Send`, `Reschedule`, and others; utility routines include `GetTID`, `GetCard`, `Lookup_Task`, and others)
`scc.a`	Include file that contains the definition of the interrupt handler table used by the routines in :A/ROSE:Examples: that makes use of SCCs

■ **Table A-1** Files on *A/ROSE 1* *continued*

File name	Description
`scc.h`	Include file that contains the definition of the interrupt handler table used by the routines in :A/ROSE:Examples: that makes use
of SCCs	
`siop.a`	Include file that contains constants used to describe hardware on the card including control register locations and some of the values that can be stored into those locations
`siop.h`	Include file that contains constants used to describe hardware on the card including control register locations and some of the values that can be stored into those locations
`timerlibrary.a`	Include file that contains the constants and definitions needed to use the timer library
`timerlibrary.h`	This include file contains the constants and definitions needed to use the timer library
FOLDER: A/ROSE 1:A/ROSE:MCP:	
`Download-lib.o`	Library containing Download and Findcard subroutines tailored to code to the MCP card
`OS.o`	Library containing A/ROSE operating system and utility routines tailored to run on the MCP card
`OSDefs.d`	Assembler symbol table file of `mrdos.a`, `os.a`, `managers.a`, and `siop.a` containing constants and macros tailored to the version of the A/ROSE operating system that runs on the MCP card
`osglue.o`	Library containing glue code and the `iopruntime` routines for programs that run on the MCP card

Files on *A/ROSE 2*

Table A-2 lists the folders and files found on the distribution disk named *A/ROSE 2* and provides a brief description of each.

■ **Table A-2** Files on *A/ROSE 2*

File name	Description
FOLDER: A/ROSE 2:Apple IPC	
`:'Apple IPC':`	Folder containing folders and files for Apple IPC
`:Forwarder:`	Folder containing the Forwarder application and files
FOLDER: A/ROSE 2:Apple IPC	
`'Apple IPC'`	Contains a driver and code to provide some of the MCP operating system features to applications running on the Macintosh II that is placed in the System Folder and the Macintosh II restarted. This file contains an INIT resource for installing the Apple IPC driver, the Apple IPC driver, the Name Manager, and the Echo Manager (ICCM is built into the Apple IPC driver)
`'Apple IPC.r'`	The `Rez` file used in the creation of the Apple IPC file that provides certain resources used within the Apple IPC file for configuration purposes. This file is provided as a quick reference to see the names and formats of those resources. Accesses to these resources are by name during initialization. These resources are not accessed by `resource ID`
`Copydriver`	The script that copies the Apple IPC file to the System Folder
`:Examples:`	Folder of example files using Apple IPC
`ipcGDefs.a`	Include file provided for debugging purposes only that contains the format of the Apple IPC driver's global data area
`ipcGDefs.h`	Include file provided for debugging purposes only that contains the format of the Apple IPC driver's global data area
`IPCGlue.o`	Library file that contains the glue interface routines necessary for using the Apple IPC driver

■ **Table A-2** Files on *A/ROSE 2* *continued*

File name	Description
FOLDER: A/ROSE 2:Apple IPC:Examples	
`'Apple IPC'`	Contains everything the Apple IPC file in folder :Apple IPC: contains plus an Echo Example task. The MakeFile shows how this file is created; the purpose of this file is to show how to add a new manager or task to the Apple IPC file
`'Apple IPC.r'`	This is the `Rez` file used in the creation of the Apple IPC file in the Examples folder (this `Rez` file is different than the `Rez` file found in the Apple IPC folder)
`:AST_ICP:`	Folder of examples for the AST_ICP card
`:DumpTrace:`	Folder for DumpTrace tool and examples
`echo.c`	The source of the Echo Example task
`echo_example`	The linked Echo Example task (the MakeFile shows how this file is created and used)
`echoglobals.a`	The source of assembler routines used within the Echo Example task
`Makefile`	Used by Make to create all of the programs and tasks within the :Examples: folder
`:MCP:`	Folder of examples for the MCP card
`name_tester`	An MPW tool designed to test the Name Manager
`name_tester.c`	The source of the test task designed to test the Name Manager
`pr_manager`	This MPW tool is a Print Manager task. The `printf` subroutine will look for a Print Manager, and request that a Print Manager print formatted text
`pr_manager.c`	This is the source of the `pr_manager` MPW tool
`RSM_File.c`	This is the source of a test task `RSM_File` which is to be dynamically downloaded to a card running MCP operating system
`RSM_tester.c`	The source of a MPW tool that dynamically downloads a task to a smart card running A/ROSE
`TestR`	This MPW tool tests the Apple IPC driver
`TestR.c`	The source of a MPW tool that tests the Apple IPC driver
`timeit`	This MPW tool measures the time required to exchange messages between itself and a Echo Manager
`timeIt.c`	The source of the MPW tool which measures the time required to exchange messages between itself and a Echo Manager
`TraceMonitor`	This MPW tool receives messages from Trace Managers and records them in a trace file
`trace_monitor.c`	The source of the TraceMonitor MPW tool

■ **Table A-2** Files on *A/ROSE 2* *continued*

File name	Description
FOLDER: A/ROSE 2:Apple IPC:Examples:AST_ICP	
`RSM_File`	An example of a module built to be dynamically downloaded to the AST-ICP card
`RSM_tester`	This MPW tool dynamically downloads a module to the AST-ICP card
FOLDER: A/ROSE 2:Apple IPC:Examples:DumpTrace	
`DumpTrace`	This MPW tool analyzes a trace file created by the TraceMonitor and dump selected messages from the trace file
`dump_16_bytes.c`	The source of one of the subroutines comprising the DumpTrace MPW tool
`dump_line.c`	The source of one of the subroutines comprising the DumpTrace MPW tool
`dump_memory.c`	The source of one of the subroutines comprising the DumpTrace MPW tool
`dump_message.c`	The source of one of the subroutines comprising the DumpTrace MPW tool
`dump_trace_file.c`	The source of one of the subroutines comprising the DumpTrace MPW tool
`init.c`	The source of one of the subroutines comprising the DumpTrace MPW tool
`is_selected.c`	The source of one of the subroutines comprising the DumpTrace MPW tool
`main.c`	The source of the main routine comprising the DumpTrace MPW tool
`Makefile`	The MakeFile used when building the DumpTrace MPW tool
FOLDER: A/ROSE 2:Apple IPC:Examples:MCP	
`RSM_File`	An example of a module built to be dynamically downloaded to the MCP card
`RSM_tester`	This MPW tool dynamically downloads a module to the MCP card
FOLDER: A/ROSE 2:Forwarder	
`FWD`	Contains the ADSP forwarder used within MacAPPC, along with an INIT resource that installs the forwarder
`fwd.h`	Include file that contains constants and definitions used by applications using the ADSP forwarder
`fwd.r`	The Rez file used in the creation of the `FWD` file (certain resources are used within the `FWD` file for configuration purposes; this file is provided as a quick reference to see the names and formats of those resources. Accesses to these resources are by name during initialization. These resources are not accessed by `resource ID`)

Appendix B **Where to Go for More Information**

In addition to the books about the Macintosh II itself, there are books on related subjects. *Table B-1* lists of reference materials that you might find helpful. ■

■ **Table B-1** List of reference material

Name	Description
Inside Macintosh, Volumes I—V	Provides a complete reference to the Macintosh Toolbox and Operating System for the original 64 KB Macintosh, Macintosh Plus (128 KB ROM), Macintosh SE, and Macintosh II (256 KB ROM)
Macintosh Programmer's Workshop (MPW) *Reference*	Describes the software programming environment for the Macintosh computer. This manual includes a combined editor and command interpreter, 68000 family assembler, linker, debugger, Macintosh ROM interfaces, resource editor, resource compiler and decompiler, and a variety of utility programs. (Version 2.0 contains complete interfaces to both the Macintosh SE and Macintosh II ROMs, improved structured macro processing from the assembler, editor markers, performance enhancements, ease-of-use features, and a variety of new commands.)
MPW C Language Manual	Describes a native Macintosh C compiler, the standard C library, Macintosh interface libraries, and offers sample programs (Version 2.0 contains full interfaces to both the Macintosh SE and Macintosh II ROMs)
MPW Assembly Language Manual	Tells you how to prepare source files to be assembled by MPW Assembler (Version 2.0 also contains interfaces to both the Macintosh SE and Macintosh II ROMs)
Designing Cards and Drivers for Macintosh II and Macintosh SE	Contains the hardware and software requirements for developing cards and drivers for the Macintosh II and the Macintosh SE (this document covers Apple's implementation of the NuBus interface in the Macintosh II and the Apple's SE-Bus interface in the Macintosh SE)
Human Interface Guidelines: The Apple Desktop Interface	Detailed guidelines for developers implementing the Macintosh user interface
Technical Introduction to the Macintosh Family	Introduction to the Macintosh software and hardware for all Macintosh computers: the original Macintosh, the Macintosh Plus, the Macintosh SE, and the Macintosh II

These documents are available to internal Apple developers through the Engineering Support Library, or to third-party developers through APDA™ (formerly, the Apple Programmer's and Developer's Association).

APDA™ provides a wide range of technical products and documentation, from Apple and other suppliers, for programmers and developers who work on Apple equipment. For information about APDA, contact

APDA
Apple Computer, Inc.
20525 Mariani Avenue, Mailstop 33-G
Cupertino, CA 95014-6299

(800) 282-APDA, or (800) 282-2732
Fax: 408-562-3971
Telex: 171-576
AppleLink: APDA

If you plan to develop hardware or software products for sale through retail channels, you can get valuable support from Apple Developer Programs. Write to

Apple Developer Programs
Apple Computer, Inc.
20525 Mariani Avenue, Mailstop 51-W
Cupertino, CA 95014-6299

In addition, you may find the references listed in *Table B-2* available through your local bookstore or computer dealer to be helpful.

- **Table B-2** Additional references

Name	Description
Motorola M68000 8-/16-/32-Bit Microprocessors Programmer's Reference Manual	Describes the latest information to aid in the completion of software systems using the M68000 family of microprocessors. This manual also covers the MC68008 8-bit data bus device, the MC68010 virtual memory processor, and the MC68012 extended virtual memory processor

Glossary

❖ *Note:* All terms in this glossary apply to the Macintosh II family of computers

address: a number used to identify a location in the computer's address space (some locations are allocated to memory, others to I/O devices)

address bus: the path along which the addresses of specific memory locations are transmitted. The width of the path determines how many addresses can be accessed (addressed) directly by the computer

address space: a range of memory locations accessible by a CPU

A/ROSE Prep: a driver and support software (library code, managers, and so forth) that handle message passing between a Macintosh application and applications running under A/ROSE on MCP-based smart cards or on another computer. A/ROSE Prep and A/ROSE perform similar functions on the main logic board and smart cards, respectively

A/ROSE: Apple Real-time Operating System Environment; the multitasking operating system for MCP-based smart cards that provides an intelligent peripheral-controller interface to the NuBus

Board sResource list: the `sResource` list that describes the board in whose declaration ROM the list resides

block mode: see run-to-block mode

blocked: the state of a task or process awaiting a message, having performed a blocking `Receive`

blocking Receive: `Receive` request where a task specifies a timeout value of greater than or equal to zero. A/ROSE suspends execution of this task and schedules another (also see non-blocking `Receive`)

bus: a path along which information is transmitted electronically within a computer

bus master: at a given time, the bus device that initiates a transaction. Also, a device with the ability to initiate a NuBus transaction by asserting the `START*` line (also see NuBus)

card: a printed circuit board connected to the bus in parallel with other boards

client: for A/ROSE and A/ROSE Prep, a software process or task that requests a service from a server task or process

code segment: area of memory for loading code that is allocated before a parent task invokes `StartTask` or `RSM_StartTask`. The code segment is automatically deallocated when the parent task issues either a `StopTask` or `RSM_StopTask`

coprocessor: any microprocessors on NuBus expansion cards; that is, any microprocessors in addition to the MC680x0 on the main logic board; coprocessors may perform system tasks, such as running alternative operating systems

cycle: one period of the NuBus clock, nominally 100 nanoseconds in duration and beginning at the rising edge

data bus: the path along which general information is transmitted within the computer. The wider the data bus, the more information can be transmitted at once. The Macintosh II, for example, has a 32-bit data bus. Thus, 32 bits of information can be transferred at a time, so that information is transferred twice as fast as in 16-bit computers (assuming equal system clock rates).

data segment: area of memory for storing global data that is allocated before a parent task invokes `StartTask` or `RSM_StartTask` for a new task. The data segment is automatically deallocated when the parent task issues either a `StopTask` or `RSM_StopTask`

declaration ROM: ROM on a NuBus slot card that contains slot manager information about the card and may also contain code or other resources

free memory pool: the total amount of memory on a card that is available for allocation either by an application using the `GetMem` primitive or by A/ROSE

free message pool: total number of messages available for allocation either by application code using the `GetMsg` primitive or by A/ROSE

gCommon: a table kept by A/ROSE on each MCP-based card that contains global information about all tasks and data structures associated with tasks running on the card

heap: the area of memory in which space is dynamically allocated and released on demand by the memory manager of the Macintosh operating system

Idle Chain: singly-linked list of small routines that are executed when when all tasks are blocked; the Idle Chain has the lowest priority of all tasks

intelligent card: see smart card

IPC: InterProcess Communication; provides message passing between cards and the main logic board, as well as with other computers on the network (also see A/ROSE Prep)

kernel: operating system code that runs in supervisor mode. In A/ROSE, the core software responsible for processing primitives, scheduling, IPC, and memory management; however, this code does *not* manage files or peripherals

local: point of reference when describing intercard or intertask communications; typically, the "local" card or task is the initial point of origin in a message-passing transaction. (also see remote)

major tick: the smallest time unit recognized by tasks running under A/ROSE (see also minor tick)

managers: tasks that carry out higher-level services on behalf of other tasks. A/ROSE managers extend the kernel to provide services not in the kernel

master: a card that initiates the addressing of a card or the main logic board across the NuBus; the card addressed is at that time acting as a slave

MCP card: the board provided with the Macintosh Coprocessor Platform that developers can use to build their own NuBus expansion card for the Macintosh II family of computers

MCP_Diagnostic: the diagnostic application provided with MCP

message: structure containing data to be passed by A/ROSE between two tasks running in a machine

message buffer: buffer or block of memory large enough to hold a message; a message buffer is allocated by application code using the `GetMsg` primitive

message ID: a statistically unique 32-bit number assigned by A/ROSE to identify each messsage buffer

message queue: a list of all messages that have been sent to a task but have not yet been received

minor tick: the smallest unit of time recognized by A/ROSE in scheduling tasks; tasks can be switched at minor ticks (see also major tick)

non-blocking Receive: `Receive` request where a task specifies a negative timeout value. A/ROSE returns control to the task immediately with either a message matching the criteria specified on the `Receive` request, or zero if no message is available. A/ROSE will not attempt to schedule another task for execution (also see blocking `Receive`)

NuBus: a synchronous bus defined by Texas Instruments that operates on a 10 MHz clock, with a full 32-bit data and address transfer. Apple's implementation of NuBus does not include parity checks, but does add interrupt lines to each of the Macintosh II NuBus slots

PAL™: an integrated circuit that implements Programmable Array Logic

parent task: any task that starts a new task

peer cards: cards that are designed to execute code that is not specialized to the card; for example, two cards that are executing cooperating processes to solve a problem

pre-emptive scheduling: in A/ROSE, a scheduling function that takes precedence when (1) a `Receive` request of a higher priority task is satisfied, the higher priority task does not necessarily execute immediately, or (2) when a task sends a message to another task on the same card, A/ROSE schedules the receiving task immediately, regardless of the priorities of the two tasks

primitive: an A/ROSE system call that provides fundamental services such as starting and stopping tasks, getting and freeing memory, getting and freeing message buffers, sending and receiving messages, changing the scheduling parameters of a task, and setting the hardware-interrupt priority level.

priority: (1) hardware priority: the status according to rank of the hardware interrupts; this status may be changed by the `SPL` primitive (a 68000 priority-level instruction); (2) task priority: the order in which a task is executed, relative to other tasks

process: an operation or function performed by the Macintosh operating system (also see blocked)

remote: point of reference when describing intercard or intertask communications; typically, the "remote" card or task is the initial destination in a message-passing transaction. Remote cards can also send messages (also see local)

run-to-block mode: a mode in which a task has control of the CPU until the task explicitly releases it. Guarantees that tasks can make uninterrupted use of the CPU.

scheduling: a function of A/ROSE that suspends one task and selects another task for immediate execution

server: for A/ROSE and A/ROSE Prep, a software process or task that provides a service to client tasks or processes

slave: a card that responds to being addressed by another card acting as a master. For example, the Macintosh II main logic board may be either master or slave. Also, a device that cannot initiate a NuBus transaction or arbitrate requests for bus mastership

slice mode: a mode in which a the operating system enables time-slicing by temporarily suspending execution of the task to allow tasks of equal or higher priority to run

slot: (1) a connector attached to the bus. A card may be inserted into any of the physical slots (the Macintosh II has six slots). (2) An area of address space allocated to a physical slot (also see slot space)

slot ID: the hexadecimal number corresponding to each card slot. Each slot ID is established by the main logic board of the Macintosh II and communicated to the card

Slot Manager: a set of Macintosh II ROM routines that enable applications to have access to declaration ROM information on slot cards (also see slot 0 managers)

slot space: an area of address space allocated to a physical slot; the upper one-sixteenth of the total NuBus address space. These addresses are of the form $Fsxx xxxx where F, s, and x are hexadecimal digits of 4 bits each. This address space is geographically divided among the NuBus slots according to the slot ID number. The slot space for each slot is 16 megabytes (also see superslot)

slot 0 managers: A/ROSE Prep manager processes, running on the Macintosh II; the Macintosh itself is sometimes referred to as the slot 0 card.

smart card: a card containing one or more processors that can work independently of the main processor of the computer. Smart cards can serve as a medium for introducing new processor technologies into a system, but most personal computer architectures require too much support from the main processor for this to happen. NuBus, however, is a notable exception, because it was designed specifically to support multiple processors, and hence, smart cards (also referred to as intelligent cards)

sResource: a software structure in the declaration ROM of a slot card. (`sResource` is analagous to, but not the same as, system resources under the Macintosh operating system.)

sResource directory: the structure in a declaration ROM that provides access to its sResource lists

sResource list: a list of characteristics of a slot resource

start parameters: data passed to a new task started by a `StartTask` or `RSM_StartTask` call by the parent task

super slot space: the large portion of memory in the range $9000 0000 through $EFFF FFFF. NuBus addresses of the form sxxx xxxx (that is, $s000 0000 through $sFFF FFFF) reference the super slot space that belongs to the card in slot s, where s is an ID digit in the range $9 through $E

supervisor mode: privilege state of the 68000 on the MCP card; the A/ROSE kernel operates in the supervisor mode, a higher state of privilege than user mode (contrast with user mode)

task: semi-independent software program code designed to accomplish specific functions that communicates by messages; this code is isolated from interfering with other tasks running at the same time; a task has its own stack space and resources

Task ID: a value that may be used to uniquely identify a task running in a machine

Tick Chain: singly-linked list of very small routines that are executed at every major tick

timeout period: the time period that a bus master waits for a non-responding slave to respond before generating a bus timeout error code

time-sliced: when the operating system temporarily suspends execution of the task to allow tasks of equal or higher priority to run. (also see slice mode.)

transaction: a complete NuBus operation, such as read or write. In the Macintosh II, a transaction consists of an address cycle, wait cycles as required by the responding card, and a data cycle. Address cycles are one clock period long and convey address and command information. Data cycles are also one clock period long and convey data and acknowledgement information

user mode: privilege state in the 68000 on the MCP card; the user mode has the lower state of privilege (contrast with supervisor mode)

utilities: A/ROSE library routines that provide for moving data, managing buffers, obtaining the operating environment, translating NuBus addresses, and registering and looking up task names.