

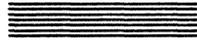


Apple.

Macintosh Coprocessor
Platform™ Developer's
Guide

Final Draft

February 20, 1989



↻ APPLE COMPUTER, INC.

This manual is copyrighted by Apple, with all rights reserved. Under the copyright laws, this manual may not be copied, in whole or in part, without the written consent of Apple Computer, Inc. This exception does not allow copies to be made for others, whether or not sold, but all of the material purchased may be sold, given or lent to another person. Under the law, copying includes translating into another language.

© Apple Computer, Inc., 1987, 1988
20525 Mariani Avenue
Cupertino, California 95014
(408) 996-1010

Apple, the Apple logo, AppleTalk, LaserWriter, and Macintosh are registered trademarks of Apple Computer, Inc. LocalTalk, the Macintosh Coprocessor Platform, MR-DOS, and MPW are trademarks of Apple Computer, Inc.

AST and AST-ICP are trademarks of AST Research, Inc.

DEC is a trademark of the Digital Equipment Corporation.

EtherTalk is a trademark of ??

Magnetic Resonance Imaging (MRI) --is this trademarked at all, or just an industry term?

NuBus is a trademark of Texas Instruments.

Open Systems Integration (OSI) - trademark? or any acknowledgement for this proposed standard

Systems Network Architecture (SNA) is a registered trademark of International Business Machines Corporation. just the initials or the whole thing?

Simultaneously published in the United States and Canada.

Contents

Figures and tables / xx

Preface / i

1 What Is MCP? / 1-1

The components of MCP / 1-2

 The MCP hardware / 1-3

 The MCP software / 1-4

 MR-DOS / 1-4

 Apple IPC / 1-6

 Developmental diagnostics / 1-6

Developing with MCP / 1-6

 Development opportunities and applications / 1-7

 Off-loading task processing / 1-8

 Parallel processing / 1-8

 Interfacing or controlling / 1-8

 Data acquisition / 1-9

 Internetworking / 1-9

 Limitations / 1-9

2 Getting Started / 2-1

Preparing to use MCP / 2-2

Installing the MCP card / 2-2

Installing MCP software / 2-5

 Installing the Apple IPC driver / 2-6

Running a sample program / 2-6

 Selecting files for the sample exercise / 2-7

 Downloading files to the card / 2-9

 Verifying the sample exercise / 2-9

Where do you go from here? / 2-12

3 Introduction to the MCP Software Interface / 3-1

What is MR-DOS? / 3-2

MR-DOS primitives / 3-2

MR-DOS utilities / 3-3

MR-DOS managers / 3-3

Echo Manager / 3-4

InterCard Communications Manager (ICCM) / 3-4

Name Manager / 3-5

Print Manager / 3-5

Remote System Manager (RSM) / 3-5

Timer Library and Timer Manager / 3-6

Trace Manager / 3-6

What is Apple IPC? / 3-7

Apple IPC driver / 3-7

Apple IPC library / 3-8

Apple IPC managers / 3-8

Functions of MCP software / 3-9

Using messages for interprocess communication / 3-9

Message structures / 3-9

Mechanisms for data transfer / 3-14

Message and status codes / 3-14

The client/server relationship / 3-15

Clients and servers running on a smart card under MR-DOS / 3-15

Clients and servers running on the Macintosh II using Apple IPC / 3-16

Using task scheduling in a multitasking environment / 3-18

Task Identifiers / 3-18

Modes in which tasks run / 3-18

Timer services / 3-19

Task scheduling / 3-19

Task initialization / 3-20

Task execution / 3-20

Task termination / 3-20

Memory management / 3-21

4 MR-DOS Primitives / 4-1

Operating system primitives / 4-2

FreeMemO / 4-3

FreeMsgO / 4-3

GetMemO / 4-4

GetMsgO / 4-4

ReceiveO / 4-5

RescheduleO / 4-6
SendO / 4-9
SplO / 4-10
StartTaskO / 4-11
StopTaskO / 4-13

5 MR-DOS Utilities / 5-1

A description of utilities / 5-3

BlockMoveO / 5-4
CopyNuBusO / 5-4
Date2SecsO / 5-5
GetBSizeO / 5-6
GetCardO / 5-6
GetDateTimeO / 5-7
GetETickO / 5-7
GetgCommonO / 5-8
GetHeapO / 5-8
GetCCTIDO / 5-9
GetNameTIDO / 5-9
GetStParmsO / 5-9
GetTCBO / 5-10
GetTickPSO / 5-10
GetTIDO / 5-10
GetTimerTIDO / 5-11
GetTraceTIDO / 5-11
GetUCountO / 5-12
IncUCountO / 5-12
IsLocalO / 5-13
Lookup_TaskO / 5-13
MapNuBusO / 5-14
Register_TaskO / 5-15
Secs2DateO / 5-16
SwapTIDO / 5-17
ToNuBusO / 5-17
TraceRegO / 5-18

6 MR-DOS Managers / 6-1

MR-DOS Managers / 6-2
Echo Manager / 6-2

- InterCard Communications Manager / 6-3
 - ICC_GETCARDS / 6-3
- Name Manager / 6-4
 - Looking up tasks / 6-7
 - NM_LOOKUP_NAME / 6-7
 - NM_LOOKUP_TID / 6-8
 - Notification of Communications Loss / 6-9
 - NM_N_SLOT_REQ / 6-9
 - NM_N_SLOT_CAN / 6-9
 - Notification of Task Termination / 6-9
 - NM_N_TASK_REQ / 6-11
 - NM_N_TASK_CAN / 6-11
 - Registering tasks / 6-11
 - NM_REG_TASK / 6-12
 - NM_UNREG_TASK / 6-12
- Printing support / 6-12
 - Print buffer request / 6-14
- Remote System Manager / 6-14
 - RSM_FreeMem / 6-15
 - RSM_GetMem / 6-15
 - RSM_StartTask / 6-16
 - RSM_StopTask / 6-16
 - Finding the Remote System Manager / 6-17
 - Loading remote tasks / 6-17
- Timer library and Timer Manager / 6-17
 - Timer library / 6-18
 - TLInitTimerO / 6-18
 - TLStartTimerO / 6-18
 - TLCancelTimerO / 6-19
 - TActiveTimerO / 6-19
 - TLReceiveO / 6-19
 - Timer Manager / 6-19
 - Active Timer Query / 6-21
 - Cancel Timeout / 6-21
 - Request One-Shot Timeout / 6-22
 - Request Periodic Timeout / 6-22
- Trace Manager / 6-23
 - Turn on tracing / 6-24
 - Turn off tracing / 6-24
 - Tracing messages / 6-24
 - DumpTrace / 6-25

7 Programming Notes for MR-DOS / 7-1

- Inter-card communications / 7-2
 - Address mapping / 7-2
 - Inter-card buffer copy / 7-3
 - Inter-card message passing / 7-3
- Interrupt handlers / 7-4
- Tick Chain / 7-6
- Idle Chain / 7-7
- Writing your own download program / 7-8
 - Findcard subroutine / 7-9
 - Download subroutine / 7-9
 - Download errors / 7-10

8 Developing Smart Card Applications / 8-1

- What you will develop / 8-2
 - Before you start / 8-2
- How to create applications using MCP / 8-3
 - Create new code / 8-3
 - Modify the main program / 8-4
- Modifying the makefile / 8-14
 - MR-DOS include files / 8-14
 - MR-DOS libraries / 8-15
 - Changes to the makefile / 8-15
- Compiling and linking your code / 8-19
- Downloading code to the MCP card / 8-20
 - Calling the Downloader tool / 8-21
 - Download errors / 8-22
- Debugging your code / 8-23

9 Apple IPC / 9-1

- The Apple IPC software / 9-2
- Installing Apple IPC / 9-3
- Using Apple IPC / 9-3
- Apple IPC services / 9-4
 - CloseQueueO / 9-5
 - CopyNuBusO / 9-5
 - FreeMsgO / 9-6
 - GetCardO / 9-6

- GetETickO / 9-7
- GetICCTIDO / 9-7
- GetIPCgO / 9-7
- GetMsgO / 9-8
- GetNameTIDO / 9-8
- GetTickPSO / 9-9
- GetTIDO / 9-9
- IsLocalO / 9-9
- KillReceiveO / 9-10
- Lookup_TaskO / 9-10
- OpenQueueO / 9-11
- ReceiveO / 9-12
 - Results returned / 9-15
- Register_TaskO / 9-16
- SendO / 9-17
- SwapTIDO / 9-18

10 Using the Forwarder with Apple IPC / 10-1

- What is the Forwarder? / 10-2
- How the Forwarder sends messages / 10-3
 - Initialization / 10-3
 - Normal processing using the Forwarder / 10-4
 - Completing communication with the Forwarder / 10-5
- Using the Forwarder / 10-6
 - Installing the Forwarder / 10-6
 - Messages used by the Forwarder / 10-6
 - MC_CLOSECONNECT / 10-7
 - MC_CLOSESERVER / 10-7
 - MC_ECHO / 10-7
 - MC_OPENSERVER / 10-8
 - MC_READDATA / 10-8
 - MC_SENDDATA / 10-9
 - Using the Forwarder on the server machine / 10-9
 - Using the Forwarder from the client machine / 10-15
 - Message transactions when using the Forwarder / 10-22
 - Errors returned by the Forwarder / 10-24

11 Troubleshooting / 11-1

What happened? / 11-2

Troubleshooting MR-DOS / 11-2

Using dumpcard / 11-3

MR-DOS crashes / 11-10

Using the load map / 11-10

Using MR-DOS error codes / 11-11

eBTHH — Bad Things Have Happened / 11-12

eCAIT — Cannot Allocate Idle Task / 11-13

eCAMS — Cannot Allocate Message Space / 11-13

eCAPR — Cannot Allocate Priority Table / 11-13

eCAPT — Cannot Allocate Process Table / 11-14

eFMSG — Attempt to Free Bad Message / 11-14

eMEMB — Attempt to Free Bad Memory Buffer / 11-15

eNPTR — No Processes to Run / 11-16

eOVFL — Stack Overflow Detected / 11-17

eSMMSG — Attempt to Send Bad Message Buffer / 11-17

eSTPI — Stop Task cannot be called from interrupt routine / 11-17

eSTTI — Start Task cannot be called from interrupt routine / 11-17

eTIMQ — Task Not in Timer Queue / 11-18

Task Not Stopped / 11-18

MR-DOS hangs / 11-19

gMajorTick is not incrementing / 11-20

Determining the cause / 11-20

gMajorTick is incrementing / 11-21

A task may be waiting on a blocking Receive request / 11-22

MR-DOS may have run out of message buffers / 11-22

A task may be running in Block Scheduling Mode / 11-24

A task may be executing in an infinite loop in Slice Scheduling Mode / 11-24

Code on the Idle Chain may be executing in an infinite loop / 11-24

Troubleshooting Apple IPC / 11-25

Apple IPC crashes / 11-27

Crashes during Macintosh II startup / 11-27

Apple IPC INIT31 — Unit Table full / 11-27

Apple IPC INIT31 — No DRVR resource in file / 11-27

Apple IPC INIT31 — Failed to open driver / 11-27

Crashes with improper parameter usage / 11-28

Apple IPC FreeMsg — Bad message pointer / 11-28

Apple IPC Send — Bad message pointer or mFrom / 11-28

Crashes during driver initialization / 11-28

- Apple IPC — Missing resource: Apple IPC entries / 11-29
- Apple IPC — Unable to get space from system heap / 11-29
- Apple IPC Name Manager — Missing aipn resource: NameManagerentries / 11-29
- IPC driver crashes during execution / 11-30
 - Apple IPC KillReceive/CloseQueue — timeout queue error / 11-30
 - Apple IPC Send — timeout queue error / 11-30
 - Apple IPC Periodic processing — timeout queue error / 11-30
 - Apple IPC Receive — timeout queue error / 11-31
 - Apple IPC Receive — Interrupt routine did blocking Receive / 11-31
- IPC Name Manager crashes during execution / 11-31
 - Name Manager Receive with Completion / 11-31
 - Name Manager Receive Request Failure / 11-31
 - Name Manager Receive Request without Completion / 11-32
- IPC glue code crashes / 11-32
- Apple IPC hangs / 11-33
 - Events causing Apple IPC hangs / 11-33
 - Macintosh II 32-bit mode debugger hang / 11-33
 - Unsatisfied blocking Receive request / 11-33
 - Examining the Apple IPC global area / 11-34
 - Finding the Apple IPC global area / 11-34

12 MCP Card Specifications / 12-1

- Introduction to the MCP card / 12-2
- Hardware description / 12-3
 - Processor / 12-3
 - ROM / 12-3
 - RAM / 12-4
 - Address map / 12-4
 - Timer / 12-5
 - Reset / 12-5
 - Interrupts / 12-5
- NuBus interface / 12-6
 - NuBus address space / 12-6
 - Acquiring the internal 68000 bus / 12-6
 - Design notes for NuBus / 12-7

13 Lists for the MCP Card / 13-1

- PAL listings / 13-2
 - PAL equation: arbitration / 13-3
 - PAL equation: bus driver / 13-4

- PAL equation: bus master / 13-5
- PAL equation: bus master control / 13-6
- PAL equation: bus slave / 13-7
- PAL equation: decode / 13-9
- PAL equation: DMA example / 13-10
- PAL equation: interrupt / 13-11
- PAL equation: RAM / 13-12
- PAL equation: RAM24 / 13-14
- Parts for the MCP card / 13-16

14 Diagnostics for the MCP Card / 14-1

- What does Apple provide? / 14-2
- Diagnostic capabilities / 14-3
- MCP card declaration ROM / 14-5
 - Power-up diagnostics / 14-5
 - 68020 primary initialization tests / 14-6
 - Data area / 14-7
 - Error codes / 14-8
- Using the MCP_Diagnostic library / 14-9

15 MCP Sequential Diagnostics / 15-1

- An overview / 15-1
 - NuBus support / 15-1
- MCP_Diagnostic main window / 15-1
- MCP menu / 15-4
 - Slot n / 15-5
 - Failure Analysis / 15-6
 - Run Script... / 15-6
 - Run Script Repeatedly... / 15-6
 - Run Script at Startup... / 15-7
 - Run Level Three Shell ... / 15-7
 - VendorMenu Item / 15-7
- Options menu / 15-7
 - Auto Run is Selected / 15-8
 - Auto Run is NotSelected / 15-8
 - Save Configuration / 15-8
 - Quit / 15-9
 - Eject and Reset / 15-9

- Debug Aids menu / 15-9
 - Stop After Pass / 15-9
 - Enable Micro Stepping / 15-10
 - Enable One Test Stepping / 15-10
 - Enable Verbose Data Logging / 15-10
 - Zero Data Log File / 15-10
 - Clear Graph / 15-11
 - Disable All Logging / 15-11
- Display menu / 15-11
 - Show Controls / 15-11
 - Testing RAM on the MCP card / 15-12
 - Testing ROM on the MCP card / 15-13
 - Testing the 68000 / 15-14
 - Testing NuBus / 15-15
 - Reading from Macintosh II system ROM / 15-16
 - Writing to Macintosh II system RAM / 15-16
 - Reading from Macintosh II system RAM / 15-16
 - MCP interprocessor tests / 15-16
 - Test and set using 68000 memory / 15-16
 - Test and set using 68020 memory / 15-16
 - Reset/timer/interrupts / 15-17
 - Level 1 timer interrupt / 15-17
 - Level 1 timer speed verification / 15-17
 - Level 2 NuBus Interrupt / 15-18
 - The MCP Card interrupts the Macintosh II 68020 / 15-18
 - Show Bits / 15-18
 - Show Data Log / 15-18
 - Show Measurement Log / 15-19
 - Show Graph / 15-19
 - Ignore Show Bits / 15-19

16 Adding to MCP / 16-1

- Adding code to the ROM / 16-2
 - The file ApplROM.a / 16-3
 - Board sResource list / 16-4
 - Application-specific driver sResource list / 16-4
 - The file ApplPowerOn.a / 16-5
 - The file ApplPrimaryInit.a / 16-5
 - The file Application.h / 16-5
 - The file ROMburn / 16-5
- Adding required resources in the ROM / 16-6

- sMemory resource list / 16-6
- sMemory resource list identifier / 16-8
- Source files for adding tests / 16-10
- Including new tests in the MCP_Diagnostic / 16-10
 - Adding menu commands to the MCP_Diagnostic / 16-11
 - Macintosh address mode compatibility / 16-11
 - Trapping bus errors / 16-12
 - The Dial routine / 16-12
- The tester script language / 16-13
 - The control section / 16-13
 - Conditional tests / 16-14
 - Examples / 16-15
 - The message section / 16-16
 - Comments / 16-16
 - Error reporting / 16-16
 - Reserved words / 16-17
 - The temporary file / 16-17
 - Script control / 16-17

17 MCP Coprocessor Diagnostics / 17-1

- What are coprocessor diagnostics? / 17-2
 - Entering third-level tests / 17-2
 - Starting third-level tests / 17-4
- Third-level menus / 17-5
 - File / 17-5
 - Quit / 17-6
 - Edit / 17-6
 - MCP / 17-6
 - Mac II Window / 17-7
 - Serial A Window / 17-7
 - Serial B Window / 17-7
 - Serial Setup / 17-8
 - Disable Verbose Messages / 17-8
- Third-level operations / 17-8
- Writing coprocessor diagnostics / 17-8
 - Creating a stack file / 17-10
- Operator commands / 17-11
 - Dumpregs / 17-12
 - Freemem / 17-12
 - Getmem / 17-13

Kill / 17-14
Readmem / 17-15
Run / 17-16
Send / 17-17
Writemem / 17-18
Buffer management / 17-19
Programmer subroutines / 17-22
 errprintf() / 17-22
 GetCards() / 17-23
 GetSlot() / 17-23
 GetTimeStamp() / 17-24
 HandleSystemTask() / 17-24
 HexToString() / 17-25
 InitMessage() / 17-25
 KillThisTask() / 17-25
 LogError() / 17-26
 printf() / 17-26
 ReadByte() / 17-26
 ReadMessage() / 17-27
 ReadWord() / 17-27
 Reply() / 17-27
 SendNextCommand() / 17-28
 StringToHex() / 17-28
 strlen(), strcpy(), and strcat() / 17-29
 TickCount() / 17-29
 WriteByte() / 17-29
 WriteMessage() / 17-29
 WriteWord() / 17-31

Appendix A Files on the MCP Distribution Disks / A-1

What this appendix tells you / A-2
Files on MR-DOS 1 / A-2
Files on MR-DOS 2 / A-8
Files on MCP Diagnostics / A-12

Appendix B Where to go for more information / B-1

What this appendix tells you / B-2

Figures and Tables

CHAPTER 1 **What is MCP?**

- Figure 1-1 Macintosh Coprocessor Platform in the Macintosh II
- Figure 1-2 The MCP card
- Table 1-1 Features of MR-DOS

CHAPTER 2 **Getting Started**

- Figure 2-1 Aligning the card
- Figure 2-2 MPW window
- Figure 2-3 Select Current Directory window

CHAPTER 3 **Introduction to the MCP Software Interface**

- Figure 3-1 Structure of MR-DOS
- Figure 3-2 Flow of information between MR-DOS and managers
- Figure 3-3 Fixed-length message structure
- Figure 3-4 Client/server transaction
- Figure 3-5 Client/server relationship for MR-DOS program modules (NuBus card-to-NuBus card)
- Figure 3-6 Client/server relationship for applications using the Apple IPC driver (Macintosh II-to-Macintosh II)
- Table 3-2 Structure for fixed-length messages
- Table 3-3 Message and status codes

CHAPTER 4 **MR-DOS Primitives**

- Table 4-1 MR-DOS primitives
- Table 4-2 Reschedule options

CHAPTER 5 **MR-DOS Utilities**

- Table 5-1 MR-DOS utilities

CHAPTER 6 **MR-DOS Managers**

- Table 6-1 MR-DOS managers
- Table 6-2 Card status
- Table 6-3 Name Manager message codes
- Table 6-4 Printf standard conversion
- Table 6-5 Printf nonstandard conversion

CHAPTER 7 Programming Notes for MR-DOS

Table 7-1 Error constants for Download

CHAPTER 8 Developing Smart Card Applications

Table 8-1 Include files

Table 8-2 Link command parameters

Table 8-3 Error constants for Download

Table 8-4 Dump area format

CHAPTER 9 Apple IIPC

Table 9-1 Apple IPC services

Table 9-2 State stable for the Receive call

Table 9-3 Errors returned

CHAPTER 10 Using the Forwarder with Apple IIPC

Figure 10-1 Messages paths using the Forwarder

Figure 10-2 Initialization process using the Forwarder

Figure 10-3 Normal processing using the Forwarder

Figure 10-4 End of processing using the Forwarder

Table 10-1 Messages used by the Forwarder

Table 10-2 Errors returned by the Forwarder

CHAPTER 11 Troubleshooting Guide

Table 11-1 Crash area forma

Table 11-2 Dumpcard cross reference

Table 11-3 Error codes for MR-DOS

Table 11-4 Error codes for Apple IPC driver

Table 11-5 Error messages from the INIT resource

Table 11-6 Error messages from the Apple IPC driver/Name Manager

CHAPTER 12 MCP Card Specifications

Figure 12-1 MCP card installed in the Macintosh II

Figure 12-2 MCP card functions

Figure 12-3 Generation of 20-MHz and 10MHz clocks

Figure 12-4 A simple NuBus slave design

Figure 12-5 Read and writing timing cycles

Table 12-1 Address map

Table 12-2 Interrupt priorities

CHAPTER 13 Lists for the MCP Card

Table 13-1 Parts lists for the MPC card

CHAPTER 14 **Diagnostics for the MCP Card**

Table 14-1	Diagnostics folders
Table 14-2	Levels of MCP diagnostics provided
Table 14-3	Data area
Table 14-4	Error codes

CHAPTER 15 **MCP Sequential Diagnostics**

Figure 15-1	MCP_Diagnostic main window
Figure 15-2	MCP menu
Figure 15-3	Warning dialog box
Figure 15-4	Menu dialog box (Run Script... command)
Figure 15-5	Dialog box (Run Script Repeatedly... command)
Figure 15-6	Options menu
Figure 15-7	Save Configuration dialog box
Figure 15-8	Debug Aids menu
Figure 15-9	Zero Data Log File dialog box
Figure 15-10	Display menu
Figure 15-11	Block 1—RAM test blocks
Figure 15-12	Block 2—ROM test blocks
Figure 15-13	Block 3—68000 test blocks
Figure 15-14	Block 4—NuBus test blocks
Figure 15-15	Block 5—Interrupt test blocks
Figure 15-16	The Data Log window
Table 15-1	Tests on-card RAM
Table 15-2	How errors are returned for RAM tests

CHAPTER 16 **Adding to MCP**

Figure 16-1	sMemory resource list for a generic MCP card
Figure 16-2	SetVar dialog box
Table 16-1	ROM:MCP files
Table 16-2	Resource list entries in the file ApplROM.a
Table 16-3	Resource list for smart cards
Table 16-4	Resources required for MR-DOS
Table 16-5	MCP_Diagnostics routine files
Table 16-6	Test description parameters

CHAPTER 17 **MCP Coprocessor Diagnostics**

Figure 17-1	Third-level diagnostics menu and dialog box
Figure 17-2	Window into a text file
Figure 17-3	File menu for third-level tests
Figure 17-4	Edit menu for third-level tests
Figure 17-5	MCP menu for third-level tests
Figure 17-6	MCP card buffer
Table 17-1	Options for third-level Dumpregs command
Table 17-2	Options for third-level Freemem command
Table 17-3	Options for third-level Getmem command
Table 17-4	Options for third-level Kill command
Table 17-5	Options for third-level Readmem command
Table 17-6	Options for third-level Run command
Table 17-7	Options for third-level Send command
Table 17-8	Options for third-level Writemem command
Table 17-9	Format of message
Table 17-10	Slots for application
Table 17-11	Returns for SendNextCommand()

APPENDIX A **Files on the MCP Distribution**

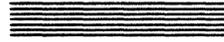
Table A-1	Files on MR-DOS 1
Table A-2	Files on MR-DOS 2
Table A-3	Files on MCP Diagnostics

APPENDIX B **Where to Go for More Information**

Table B-1	List of reference material
Table B-2	Additional references



Preface



About This Manual

What this guide tells you

This guide is intended to inform and assist you in your efforts to create an interface to the Macintosh® II bus. Developers may be within Apple Computer, Inc., as well as third-party developers (such as a VAR channel or a national account) working under a licensing agreement.

What you should know

You should be familiar with the Macintosh II computer and NuBus™. Refer to Appendix B for information on developer tools and reference documents that may facilitate your development efforts.

The Macintosh Coprocessor Platform™ (MCP) is intended to support 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

This section provides a road map to information on the various components of the Macintosh Coprocessor Platform.

To find out about:

General information on MCP

What makes up the Macintosh Coprocessor Platform

Applications or potential uses of MCP

Installing the MCP card and running a sample program

Specific information on MCP software

MR-DOS™ and Apple IPC software in the Macintosh II

Task scheduling in the operating system

Interprocess communication between tasks or processes on the Macintosh II and tasks on the MCP card

Look in:

Part I — Getting Started

Chapter 1, "What is MCP?"

Chapter 1

Chapter 2, "Getting Started"

Part II — Software Development

Chapter 3, "An Introduction to the MCP Software"

Chapter 3

Chapter 3 for general information (for additional information, see Chapter 9, "Apple IPC")

Fundamental services of the MR-DOS operating system	Chapter 4, "MR-DOS Primitives"
Library routines available to tasks in your application	Chapter 5, "MR-DOS Utilities"
Operating-system managers that provide services to other tasks	Chapter 6, "MR-DOS Managers"
Peculiarities of MR-DOS and programming notes (with examples of code)	Chapter 7, "Programming Notes for MR-DOS"
How to develop applications using MCP software (with examples of code)	Chapter 8, "Developing MCP Applications"
MR-DOS services provided on the Macintosh II	Chapter 9, "Apple IPC"
Forwarding data on an AppleTalk® network system using Apple IPC	Chapter 10, "Using the Forwarder with Apple IPC"
Troubleshooting MCP software	Chapter 11, "Troubleshooting Guide"
Information about the MCP card and NuBus™	Part III — Hardware Development
MCP card specifications and information accessing the NuBus	Chapter 12, "MCP Card on Specifications"
PAL listings and parts lists	Chapter 13, "Lists for the MCP Card"
Information about diagnostics	Part IV — MCP Diagnostics
The diagnostics provided for development of the MCP Card	Chapter 14, "Diagnostics for the MCP Card"
Using MCP diagnostic software	Chapter 15, "MCP Sequential Diagnostics"
How to customize diagnostics provided for your application-specific card and add code to the ROM	Chapter 16, "Adding to MCP"
Creating new diagnostics for your card	Chapter 17, "MCP Coprocessor Diagnostics"

- a NuBus-compatible Macintosh computer running System 6.0.2
- 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 Development", for more information.

Important safety instructions

You are almost ready to plug in your Macintosh II and get started, but first read these important safety instructions.

Warning

This equipment is intended to be electrically grounded.

Your Macintosh II is equipped with a three-wire grounding plug—a plug that has a third (grounding) pin. This plug will fit only a grounded AC outlet. This is a safety feature.

If you are unable to insert the plug into the outlet, contact a licensed electrician to replace the outlet with a properly grounded outlet.

Do not defeat the purpose of the grounding plug!

For your own safety and that of your equipment, always take the following precautions:

Be sure the power plug is disconnected (disconnect by pulling the plug, not the cord):

- whenever you remove the cover and as long as the cover is off
- if the power cord or plug becomes frayed or otherwise damaged
- if you spill anything into the case
- if your Macintosh II is exposed to rain or any other excess moisture
- if your Macintosh II has been dropped or if the case has been otherwise damaged
- if you suspect that your Macintosh II needs service or repair
- whenever you clean the case (use only the recommended procedure described below)

Be sure that you always do the following:

- Keep your Macintosh II, the MCP card, and distribution disks away from sources of liquids, such as wash basins, bathtubs, and shower stalls.
- Protect your equipment and materials from dampness or wet weather, such as rain and snow.

- Read all the installation instructions carefully before you plug your Macintosh II into a wall socket.
- Keep these instructions handy for reference by you and others.
- Follow all instructions and warnings dealing with your system.

Warning

Electrical equipment may be hazardous if misused. Operation of this product, or similar products, must always be supervised by an adult. Do not allow children access to the interior of any electrical product and do not permit them to handle any cables.

To clean the case, do the following:

1. Disconnect the power plug. (Pull the plug, not the cord.)
2. Wipe the surfaces of your Macintosh II lightly with a clean, soft cloth dampened with water.

Conventions

This section provides general information on the conventions used in printing 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
- the name of a file provided on the MCP distribution disks

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

MR-DOS 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 words mark special messages 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.

Terms

This document refers to **processes** on the Macintosh II computer, and **tasks** under MR-DOS and Apple IPC. 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 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.



Part I



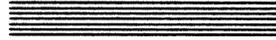
Getting Started with MCP

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-in 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.

Technical information about the components of MCP is provided in this guide, along with a discussion of potential applications. Refer to Appendix A for a description of associated development tools, documents, and references.

The components of MCP

The Macintosh Coprocessor Platform is made up of hardware, software, and developmental diagnostic software, provided as follows:

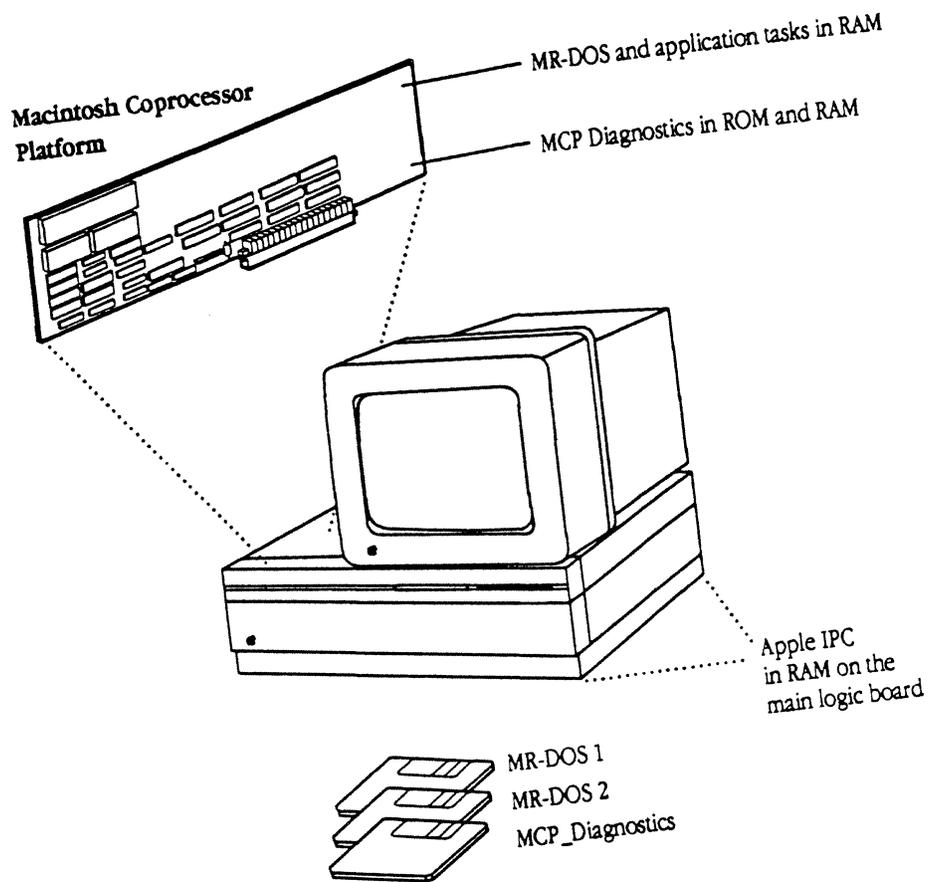
- hardware: the **MCP card**, an intelligent NuBus prototype card (such cards may be referred to as **smart cards**)
- software: two distribution disks (labeled MR-DOS 1 and MR-DOS 2) that includes **MR-DOS** (Minimal, Real-time, Distributed Operating System) and **Apple IPC** (InterProcess Communication)

MR-DOS 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 II.

Apple IPC includes a driver and support software installed in the Macintosh II computer. Apple IPC allows Macintosh applications to communicate with an application running under MR-DOS on the MCP card or 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 you develop

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



2/9/89

Fig. 1-1 -COMP (L1)
MCP Developer's Guide
Apple Computer, Inc.
JOYCE ZAVARRO
Illustrator 88
GEORGE M. VRANA

MSC NNNN
ART: NN x 17 pi
20.5 pi text to FN b/b

Figure 1-1
Macintosh Coprocessor Platform in the Macintosh II

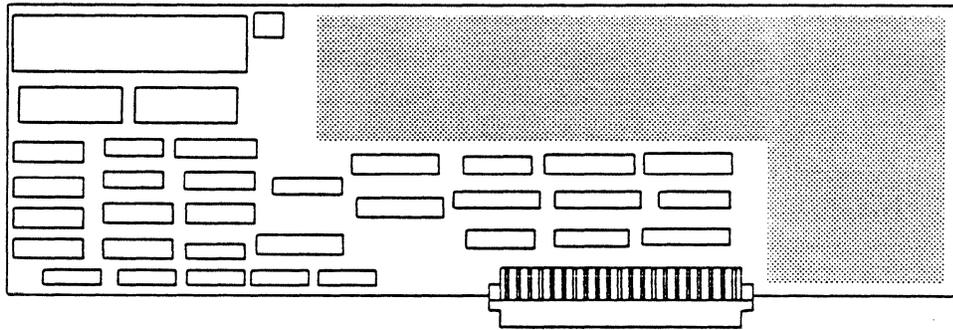
You can customize each of these components, which are described in this chapter, for the particular application or product you want to develop. For more detailed information, refer to Part II on Software Development, Part III on Hardware Development, and Part IV on Developmental Diagnostics.

The MCP hardware

With approximately 26 square inches of space available, the MCP card is intended as a vehicle for creating a prototype of the features and interface required for your product or application. Figure 1-2 shows the layout of the MCP card; shading indicates the primary area available for development.

MSC NNNN
ART: NN x 17 pi
20.5 pi text to FN b/b

Figure 1-2
The MCP card



*Fig. 1-2 -COMP (L2)
MCP Developer's Guide
Apple Computer, Inc.
JOYCE ZAVARRO
Illustrator 88
GEORGE M. VRANA*

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 Ring 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 II main logic board. The MCP card acts as a "slot device" to the Macintosh II operating system, freeing the processor on the Macintosh II 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 MR-DOS, Apple IPC, and support software (include files, source code examples, and other development software tools). MCP software was created to take advantage of the common design features of the MCP card by providing common software services to smart card application programs.

The code for MR-DOS and Apple IPC 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 II computer.

❖ *Note:* To assist in development efforts, MCP software is released in versions that support two smart cards: the MCP card and the AST-ICP card.

MR-DOS

MR-DOS provides the operating system and core software services required by MCP cards for on-board applications software. The design of MR-DOS 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 MR-DOS

Feature	Description
Configurability	For maximum flexibility in meeting the needs of a variety of products, large parts of MR-DOS are configurable. MR-DOS code that supports services not required by an application need not be loaded onto the MCP card. To complement configurability, the MR-DOS kernel is as small as possible.
Inter-card 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. MR-DOS allows dynamic name-binding of tasks to support interprocess communication.
Multitasking	Multiple independent tasks share the CPU on the smart card, under control of MR-DOS. 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. MR-DOS 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 MR-DOS and the services it provides.

Apple IPC

Apple IPC is composed of:

- a driver that runs under the Macintosh operating system
- Apple IPC interface code
- library routines (in the file `IPCglue.o`)
- associated support code, including the Apple IPC Name Manager and Apple IPC Echo Manager

The Apple IPC driver handles all message passing (interprocess communication) between processes on the Macintosh II operating system and MCP card tasks on the NuBus.

Periodically, Apple IPC 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 II processes.

- ❖ *Note:* Since the Macintosh II 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 Apple IPC and the services it provides.

Developmental diagnostics

Developmental diagnostics are provided in the form of both firmware and software. The firmware is provided in the declaration ROM on the MCP card, and the software is provided on the third MCP distribution disks.

These diagnostics are being provided solely as a framework for test verification of board designs. You can use the basic tests provided on the distribution disk, or customize the diagnostics for the particular board you are developing. Refer to Part IV, "MCP Diagnostics", for more detailed information.

Developing with MCP

MCP provides hardware and software to assist you in creating

- an application-specific smart card
- Macintosh II application software that uses Apple IPC for communication with tasks on the card
- software that executes under MR-DOS 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 MR-DOS and Apple IPC, as well as all of the support software.

You will also need a Macintosh II computer with one or more smart cards in the expansion slots. You could conceivably create applications on a Macintosh II computer without smart cards installed, and then port it to a Macintosh II 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 a MR-DOS or Apple IPC application; refer to Chapter 8
- how to create interrupt handlers; refer to Chapter 9
- how to send data directly to another card; refer to Chapter 5

A few development opportunities and potential applications are discussed in the next section.

Development opportunities and applications

The communications and networking strategy of Apple Computer is to integrate the Macintosh II 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 standard Open Systems Integration (OSI).

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

- offload 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 II computer for the application, depending on how you customize the interface on the card. It is possible to create MCP card applications which, once downloaded, have no dependence on the Macintosh II operating system.

Any application or environment that requires the performance of a Macintosh II 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 II (commonly referred to as the motherboard) and have MR-DOS 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 II 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 II environment, using the power of the Macintosh II to control devices, collect data, or perform some type of analysis. In this situation, the Macintosh II 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 connection on the MCP card, you could connect a drive from the Macintosh II 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 II

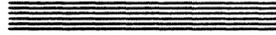
Limitations

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 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 computer, but have not yet installed any MCP software or hardware.

Preparing to use MCP

Before you proceed, follow these steps to prepare for this exercise.

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

Two of the MCP distribution disks contain source code and programming examples you will need for application software development and this exercise; these disks include MR-DOS, Apple IPC, and the support software for both. The third disk contains source code and the MCP diagnostics program you will need to create diagnostic software.

- ❖ *Note:* Please be sure to follow instructions in the section in this chapter, "Installing MCP software" 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 this exercise but for all software development efforts.

For a complete guide to the folders and files included on the MCP distribution disk, refer to Appendix A, "Development Tools and Resources". (This chapter simply identifies the folders and files you will need for this exercise.)

Now follow the instructions provided in the next section 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 II. If you are not familiar with installing cards, refer to the owner's guide for your Macintosh II 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 II.

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.

Warning

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.**

Warning

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.

Warning

It's OK to touch the power supply if you've just unpacked it. 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 fourth to the right of the video card (slot D).

- ❖ *Note:* 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 purposes of this exercise.

- Remove the expansion cover shield behind the expansion slot you plan to use by lifting up until the shield is free of the guide and pin.
- Push out the plastic hole cover that lines up with the slot you plan to use.

2. Insert the MCP card into the expansion slot.

- Being careful not to touch the pins on the bottom of the card, pick up the MCP card by the top of the metal bracket and the top of the card's other end.
- 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. 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.

MSC NNNN
ART: NN x 17 pi
20.5 pi text to FN b/b

Figure 2-1
Aligning the card

- 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.

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.

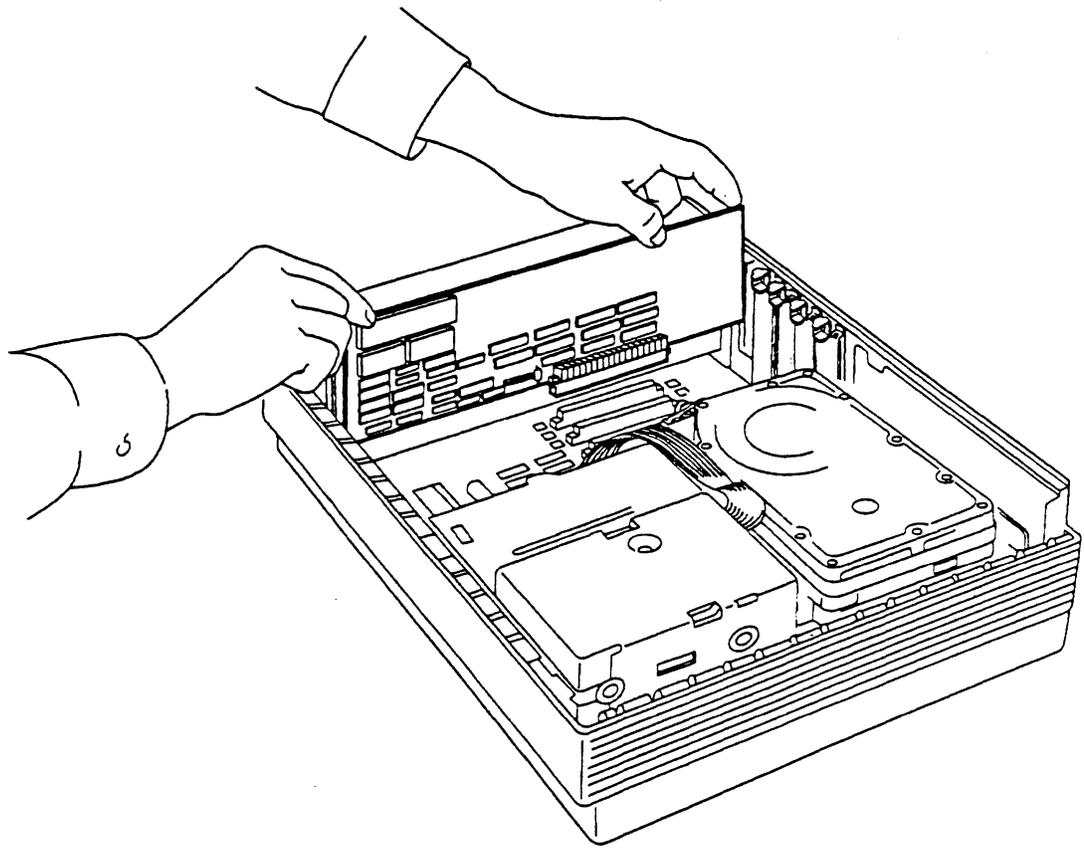
3. If you have purchased other peripheral devices that require cards, install them now.

You can use this same method for installing all expansion cards in your Macintosh II 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 II 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 II 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.



*Fig. 2-1 -COMP (L11)
MCP Developer's Guide
Apple Computer, Inc.
JOYCE ZAVARRO
Illustrator 88
GEORGE M. VRANA*

❖ *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 II.

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 II for important instructions about SCSI terminators.

Once you are satisfied that everything is connected properly, arrange the Macintosh II 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.

Warning

Always keep your computer's main unit flat, sitting on its rubber feet. Standing the main unit on edge defeats the cooling design and is likely to make your computer overheat. A vertical position may eventually damage the main unit.

Installing MCP software

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

1. **Create a new folder called MCP Software on your Macintosh II desktop.**
2. **Copy the contents of the 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 MR-DOS, 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 MR-DOS folder and Apple IPC folder must be at same level within the new folder you just created, because certain items within the Apple IPC file use data in the `include` files in the MR-DOS folder.

Installing the Apple IPC driver

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

1. Select the Apple IPC folder within the new folder you created on the Macintosh II desktop.
 2. Within the Apple IPC folder, open the Examples folder and select the Apple IPC file.
 3. Copy the Apple IPC file into the System Folder of the Macintosh II.
- ❖ *Note:* You can copy the file in one step by holding down the Option key while dragging the Apple IPC file into the System Folder.
4. Reboot the Macintosh II.

The Apple IPC driver is loaded into the system heap during system startup by an INIT31 resource within the Apple IPC file.

Running a sample program

This section describes how to run a sample program that shows the features and functions of the MR-DOS 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.

SCREEN SHOT

MSC NNNN
ART: NN x 17 pi
20.5 pi text to FN b/b

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.

MSC NNNN
ART: NN x 17 pi
20.5 pi text to FN b/b

SCREEN SHOT

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 MR-DOS.**
- 4. Open the folder named Examples.**

Figure 2-2

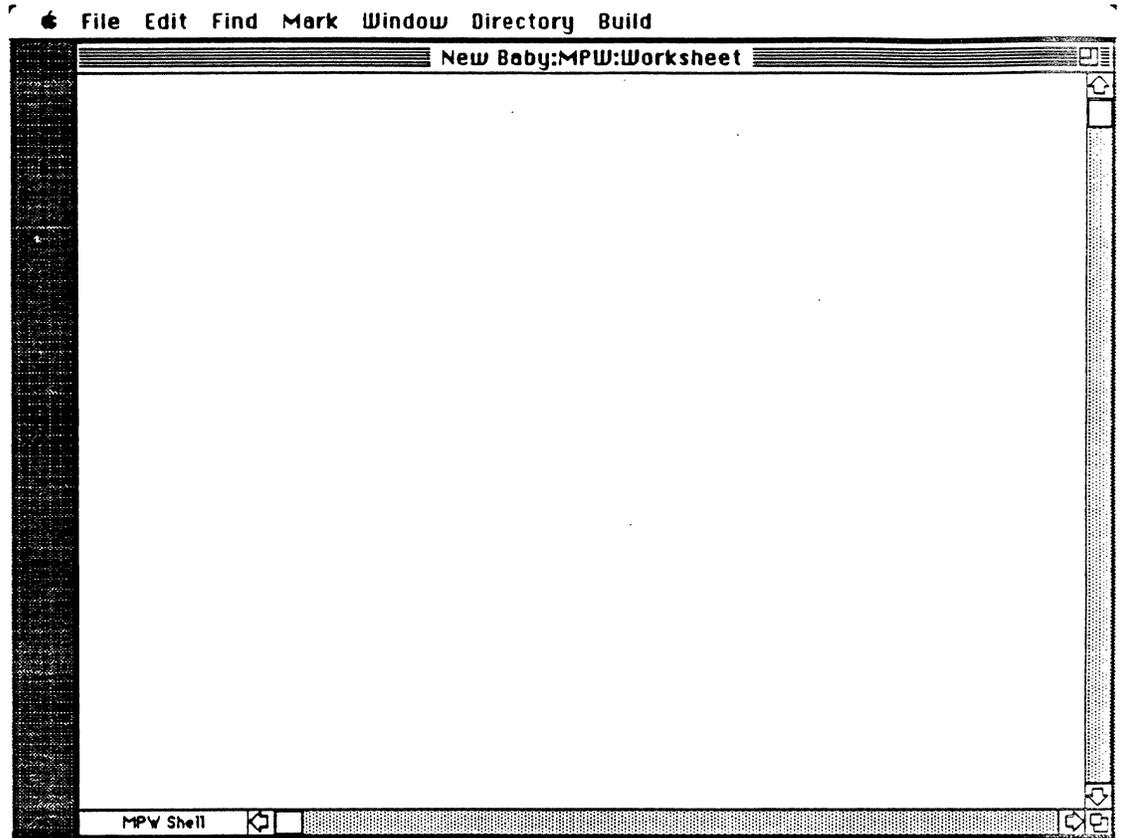
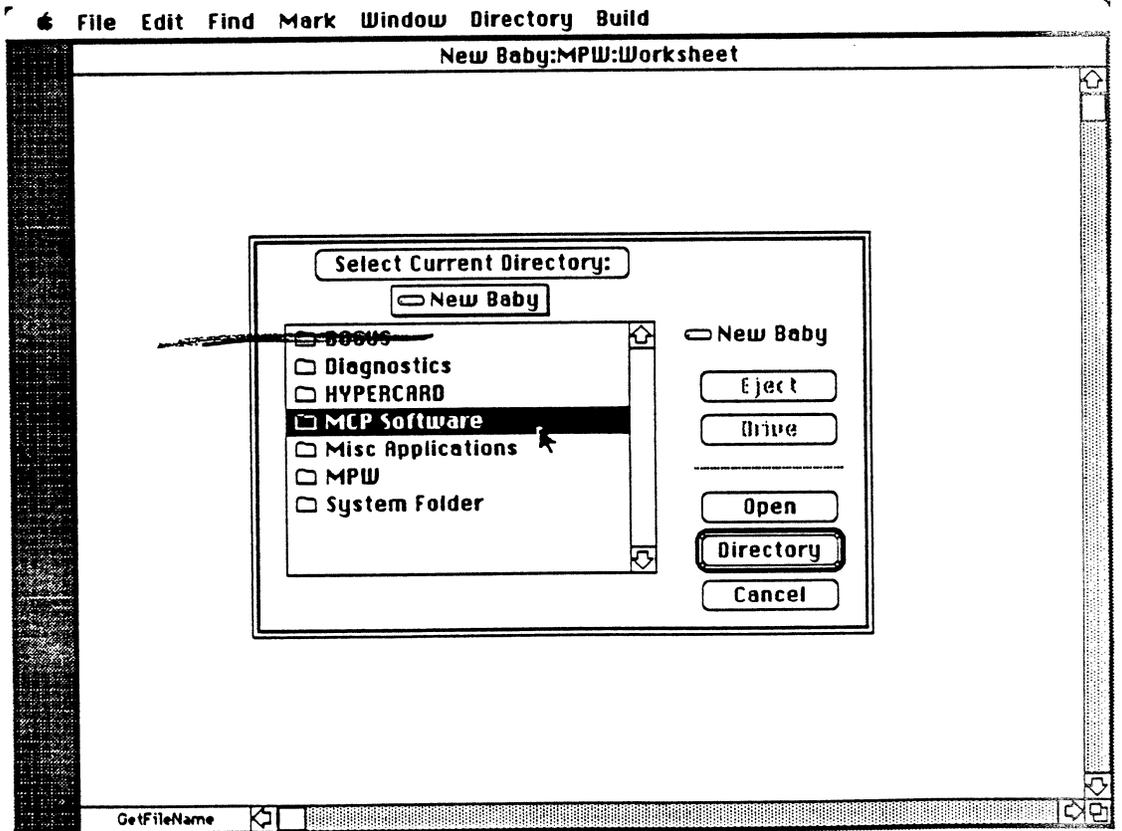


Figure 2-3



5. Select the folder named MCP.**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:MR-DOS:Examples:MCP:'  
-----
```

where: `directory` is the command you entered

`'New Baby:MCP Software:MR-DOS:Examples:MCP:'` is the pathname to the folder

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

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

```
-----  
files  
Download  
dumpcard  
echo.c.o  
L3MMSVP.a.o  
L3MMSVP.c.o  
L3MMSVPClient.c.o  
map  
name_tester.c.o  
osmain.c.o  
ossccint.a.o  
pr_manager.c.o  
printf.c.o  
start  
timeIt.c.o  
timer_tester.c.o  
trace_manager.c.o  
xref  
-----
```

For this exercise, you will use the files named `download` and `start`. The `download` file contains an MPW tool that loads code from MR-DOS 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 the file, enter both the command name and the name of the sample file, as follows:

```
download start
```

The `start` file is now running with the MR-DOS operating system on the MCP smart card in your Macintosh II. 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 provided on the MCP distribution disk called the print manager (`pr_manager`), 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 Apple IPC, 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:Apple IPC:Examples:'
```

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

You should see the following listing on the screen:

```
files
:AST_ICP:
:DumpTrace:
:MCP:
'Apple IPC'
'Apple IPC.r'
echo.c
echo_example
echoglobals.a
Makefile
name_tester
name_tester.c
pr_manager
pr_manager.c
RSM_File.c
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 MR-DOS

`TID=4` is the Task Identifier (TID) assigned to that task by MR-DOS

`b00000n` 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.
```

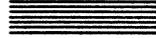
You can direct this output as you would do anything else in MPW, such as saving it to a temporary file for printing later.

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 MR-DOS and Apple IPC; Part III, "Hardware Development", provides information on hardware development; and Part IV, "MCP Diagnostics", provides information on customizing development diagnostics.



Part II



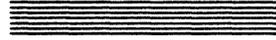
Software Development

Part II, Software Development, provides

- an introduction to and an overview of MR-DOS and Apple IPC
- definitions of operating system primitives, utilities, and managers for MR-DOS and Apple IPC, along with examples in both assembly language and C
- information on how to use the operating system
- an exercise to modify standard MCP files to build an application program
- programming guidelines and notes for MR-DOS, with program listings for selected examples
- a troubleshooting section for crashes and hangs with either MR-DOS or Apple IPC



Chapter 3



Introduction to the MCP Software Interface

Software for the Macintosh Coprocessor Platform includes MR-DOS, Apple IPC, 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.

Some of the specific concerns you may have in developing your own application may include how to:

- create a MR-DOS or Apple IPC application; refer to Chapter 8
- create interrupt handlers; refer to Chapter 9
- send data directly to another card; refer to Chapter 5

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

What is MR-DOS?

MR-DOS (Minimal, Real-time, Distributed Operating System) is a multitasking operating system for smart card devices, such as the MCP card, and provides an intelligent peripheral-controller interface to NuBus.

MR-DOS 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; MR-DOS makes minimal assumptions about how things operate. MR-DOS provides basic support services to tasks through system calls (**primitives**) and library routines (**utilities**).

MR-DOS primitives

A primitive is a MR-DOS 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 MR-DOS primitives.

MR-DOS 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 MR-DOS 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 MR-DOS utilities.

MR-DOS managers

Managers are tasks that carry out higher-level services on behalf of other tasks. MR-DOS managers extend the kernel to provide services that are not in the kernel, but are useful for all users of the MR-DOS 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 MR-DOS kernel, primitives, utilities, and managers.

MSC NNNN
ART: NN x 17 pi
20.5 pi text to FN b/b

Figure 3-1 Structure of MR-DOS

Figure 3-2 illustrates the flow of information between MR-DOS and these managers on an MCP card.

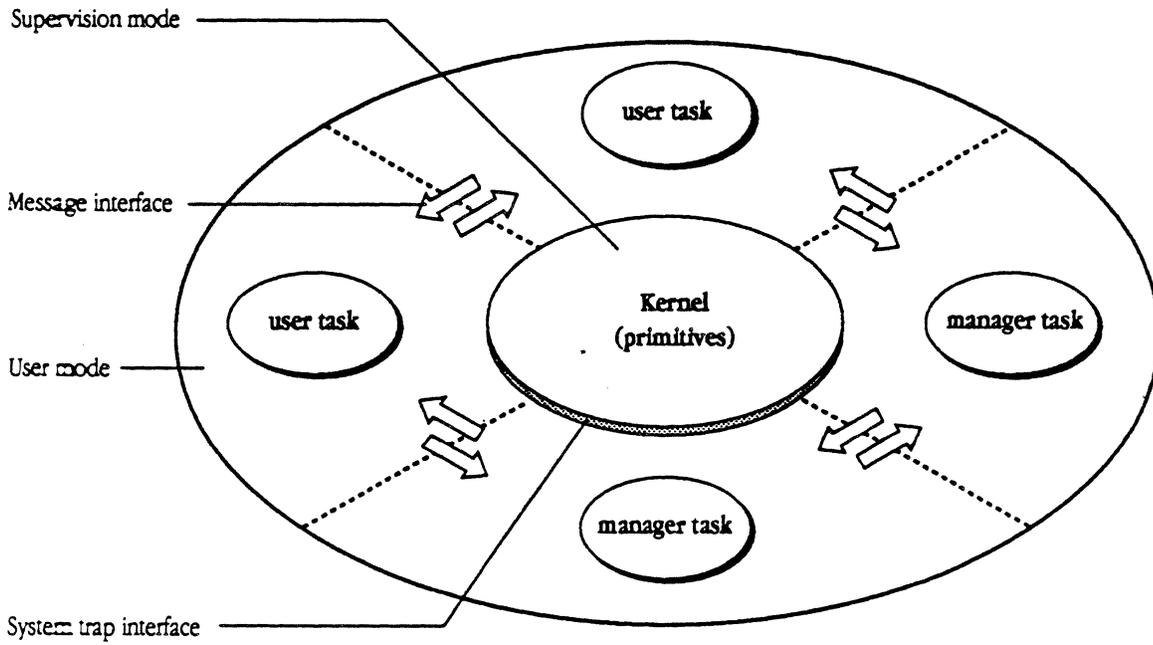


Fig. 3-1 -COMP (L3)
 MCP Developer's Guide
 Apple Computer, Inc.
 JOYCE ZAVARRO
 Illustrator 88
 GEORGE M. VRANA

MSC NNNN
ART: NN x 17 pi
20.5 pi text to FN b/b

Figure 3-2
Flow of Information between MR-DOS and managers

This section provides a brief description for each of the MR-DOS 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. MR-DOS delivers any messages addressed off-card to Apple IPC or ICCM. ICCM forwards the message to a peer ICCM on the destination smart card 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.

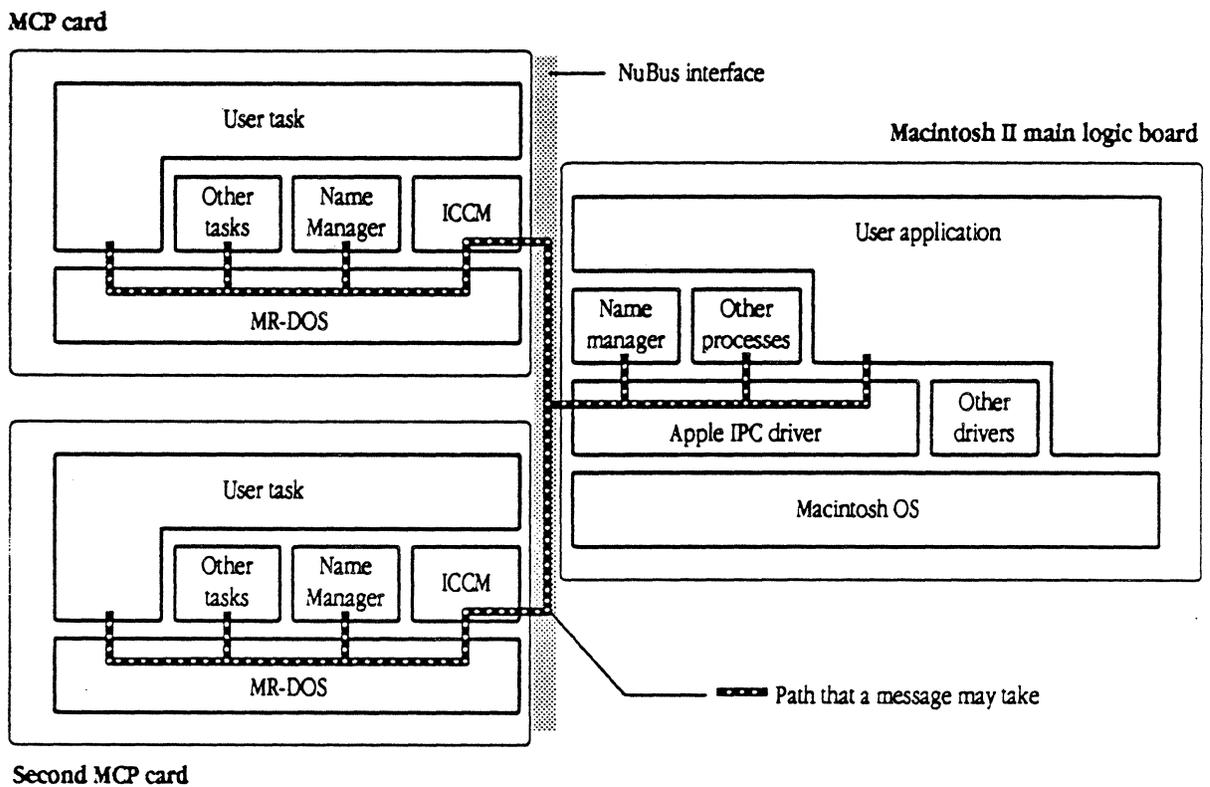


Fig. 3-2 -COMP (LA)
 MCP Developer's Guide
 Apple Computer, Inc.
 JOYCE ZAVARRO
 Illustrator 88
 GEORGE M. VRANA

Name Manager

The Name Manager allows user programs to find the task IDs of other user programs, given the names of those programs.

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 II 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 II or out 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 MR-DOS 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 programs 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 MR-DOS software for historical purposes.

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 Apple IPC?

Apple IPC (InterProcess Communication) is a combination of a driver and support software found in the Apple IPC file in the Apple IPC folder on the MCP distribution disk.

Apple IPC provides message-passing and naming services for communications from the Macintosh II to other tasks on the Macintosh II and to tasks on smart cards.

Interprocess communication is accomplished through messages that are fixed-size but flexibly formatted. (Apple IPC is similar to the InterCard Communications Manager on MR-DOS.)

❖ *Note:* This document refers to *processes* on the Macintosh II, and *tasks* under MR-DOS and Apple IPC.

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

Apple IPC driver

Apple IPC services are handled by the **Apple IPC driver**, which handles all message-passing between processes on the Macintosh II operating system and MR-DOS tasks on the smart card over the NuBus. Using calls to the Apple IPC driver, the Macintosh II process sends messages to and receives messages from tasks on the smart card processes and on the Macintosh II. In addition, Apple IPC allows communication between two or more processes running on the same Macintosh II main logic board.

The Apple IPC file is placed 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 II, much as the MR-DOS ICCM does. For each valid ICCM communication area found, the driver stores the address of the Apple IPC communication area in a vector in the ICCM's communication area.

Periodically, Apple IPC 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 II processes.

Apple IPC library

The interface between a Macintosh application and the Apple IPC driver is made through the object routines, or glue code, in the **Apple IPC 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 Apple IPC library provides access to many of the same utilities as provided by MR-DOS, such as moving data, obtaining the operating environment, and registering and looking up task names through the Apple IPC Name Manager. These routines are located in the file Apple IPC:IPCGLue.o on the MCP distribution disks. (All of these routines use the C calling sequence.)

Apple IPC managers

The managers for Apple IPC are the Echo Manager and the Name Manager. These Apple IPC managers perform functions identical to and have the same message interface as those of their MR-DOS counterparts; minor differences are due to the slightly different interface to Apple IPC.

The Apple IPC managers are processes that carry out higher-level services on behalf of applications on the Macintosh II computer. These managers are often referred to as **slot 0 managers**, and the Macintosh 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 II (part of the Macintosh operating system).

Functions of MCP software

The operation of MCP software is described in terms of the following functions:

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

Using messages for interprocess communication

Messages are the fundamental means for communication between tasks for MR-DOS and Apple IPC. 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.

Message structures

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

- 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 user
- three long words of data for the receiver
- three long words of data that should be returned untouched by the receiver in a response
- a pointer to a data buffer
- the size of the data buffer
- a message identifier (ID)
- the message priority
- the 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 MR-DOS and Apple IPC.

MSC NNNN
 ART: NN x 17 pi
 20.5 pi text to FN b/b

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.

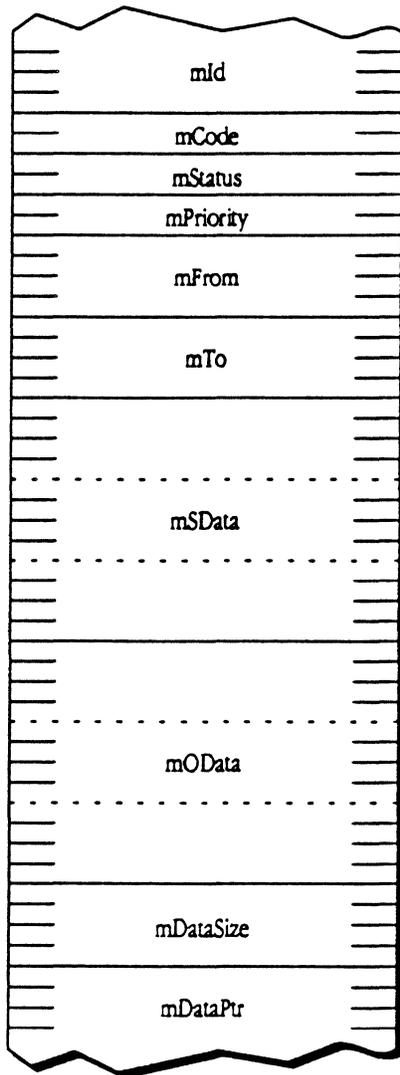


Fig. 3-3 -COMP (LS)
 MCP Developer's Guide
 Apple Computer, Inc.
 JOYCE ZAVARRO
 Illustrator 88
 GEORGE M. VRANA

Table 3-1
Structure for fixed-length messages

Field Name	Field Size	Description/Usage
<code>mId</code>	long	a statistically-unique, 32-bit number to identify the message, used when a message is obtained from MR-DOS or the Apple IPC driver by way of a <code>GetMsg()</code> request
<code>mCode</code>	short	<p>a 16-bit message code understood only by the sender and receiver of a message</p> <p>By convention, an even <code>mCode</code> is a request message, and an odd <code>mCode</code> is a reply message. You can find examples of this convention in the files <code>:MR-DOS:includes:managers.a</code> and <code>:MR-DOS:includes:managers.h</code>. For example, the ICCM request code <code>ICC_GETCARDS(150)</code> is even; the ICCM reply code <code>ICC_GETCARDS+1(151)</code> is odd. The Name Manager request code <code>NM_REG_TASK(100)</code> is even; the Name Manager reply code <code>NM_REG_TASK+1</code> is odd.</p> <p>The MR-DOS operating system, the Apple IPC driver, the managers (Name Manager, ICCM, and others) set the high bit of the <code>mCode</code> in a message if the <code>mCode</code> is not recognized or the message is undeliverable. The file <code>managers.a</code> and the file <code>managers.h</code> in the folder <code>:MR-DOS:includes:</code> list the <code>mCodes</code> known by MR-DOS, the Apple IPC driver, and the managers.</p>
<code>mStatus</code>	short	a 16-bit status code, with the upper 8 bits of <code>mStatus</code> designated as a MR-DOS system status code and the lower 8 bits of <code>mStatus</code> designated as a user status code. The <code>mStatus</code> values used by MR-DOS, Apple IPC, and the managers are found in the files <code>managers.a</code> and <code>managers.h</code> in the folder <code>:MR-DOS:includes:</code> .

For any message that is undeliverable, MR-DOS and Apple IPC change the entire `mStatus` word to a value of \$8000. If a message with `mStatus` already set to \$8000 is found to be undeliverable, MR-DOS and Apple IPC discard the message.

<code>mPriority</code>	unsigned short	a 16-bit unsigned word representing the priority of the message (0 is the lowest priority).
<code>mFrom</code>	long	<p>a source address (the task that sent the message)</p> <p>By convention, <code>mFrom</code> is the TaskID (TID) of the task sending the message. MR-DOS automatically fills in the <code>mFrom</code> field to that of the current TID when a message is obtained by a <code>GetMsg()</code> request. A task receiving a message should swap the <code>mFrom</code> and <code>mTo</code> fields before sending a message in reply.</p> <p>To declare the TID number, use <code>tid_type</code> <code>TYPEDEF</code> 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.</p>
<code>mTo</code>	long	<p>a destination address (the task to which the message is directed)</p> <p>The <code>mTo</code> field is the TaskID (TID) of the task to which you want to send a message. This field <i>must</i> be filled in before doing a <code>Send</code> request.</p> <p>To declare the TID number, use <code>tid_type</code> <code>TYPEDEF</code>. 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.</p>
<code>mSData</code>	3 long words	12 bytes of data defined by the sender, associated with the message, that should be returned untouched by the receiver in a response. This field contains internal context information meaningful only to the requesting task.

By convention within MR-DOS, 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`.

<code>mODaTa</code>	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.
<code>mDaTaSiZe</code>	long	the size of an associated data buffer pointed to by <code>mDaTaPtR</code> . This size is in 8-bit bytes.
<code>mDaTaPtR</code>	long	a pointer to an associated data buffer. MR-DOS assumes that this pointer is the address of the associated data buffer as seen from the Macintosh II NuBus. For example, suppose there is an associated data buffer on a card in slot C at address 1234; the <code>mDaTaPtR</code> should have the value FC001234, which is the NuBus address.

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

1. The message must fit any match criteria that was specified on 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 task's private use within the messaging mechanism.

❖ *Note:* This is a convention; it is not enforced by the MR-DOS operating system.

- a data buffer

The third mechanism involves passing a data buffer address and its size (that is, the length in bytes) in the message to the receiving process 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 16-bit accesses or less (for example, accessing the NuBus using the AST-ICP card).

In addition, the buffer address must be mapped to a local address with the function `MapNuBus`, described in Chapter 12. `MapNuBus` sets up any required latch registers on hardware that requires it, such as on 68000-based cards, and returns the local address to be used for the access. The operating system automatically saves and restores the address mapping for each task.

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 <ul style="list-style-type: none"> • use an even number to request services • use an odd number for replies
mStatus	16-bit	message status field <ul style="list-style-type: none"> • use the upper 8 bits 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 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 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 re-utilizes the message buffer for an outgoing message, or returns it to the message buffer pool using a `FreeMsg()` call.

Figure 3-4 illustrates the normal sequence of actions between clients and servers. This sequence is similar for clients and servers that run either on the MCP card under MR-DOS or on a Macintosh II using the Apple IPC driver. The client can be on a different slot than the server; that is, one could be on one MCP card, and the other could be on a different MCP card or slot 0 (the Macintosh II, for example).

MSC NNNN
ART: NN x 17 pi
20.5 pi text to FN b/b

Figure 3-4
Client/server relationship

Clients and servers running on a smart card under MR-DOS

This section provides an example of a client and server running on a smart card under MR-DOS. You can find the source code for this example in the folder `:MR-DOS: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 `:MR-DOS:Examples` for making the `echo.c` and `timeit.c` examples.)

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

Typical client/server transaction

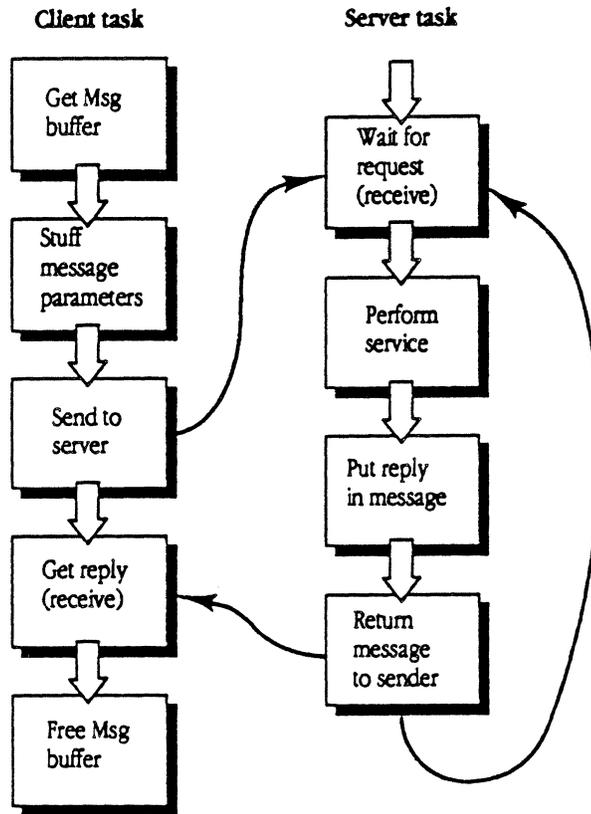


Fig. 3-4 (L6)
MCP Developer's Guide
Apple Computer, Inc.
JOYCE ZAVARRO
Illustrator 88
GEORGE M. VRANA

A 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, 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` request 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` request to send the message buffer to the client. The server next issues a `Receive` request to wait for another message from a client.

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

Figure 3-5 illustrates this process for program modules containing MR-DOS running on the MCP card.

MSC NNNN
 ART: NN x 17 pi
 20.5 pi text to FN b/b

Figure 3-5
 Client/server relationship for MR-DOS program modules

Clients and servers running on the Macintosh II using Apple IPC

The sequence of actions needed for clients and servers running on the Macintosh II using Apple IPC is similar to that described above. This section also describes some of the differences between an application program running on the Macintosh II and program modules running under MR-DOS on the MCP card.

A server and client process using the Apple IPC driver on the Macintosh II is different from a server and client process running under MR-DOS due to the differences between MR-DOS and the Macintosh II operating system; that is, MR-DOS is a multitasking operating system, and the Macintosh II 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 `:MR-DOS:Apple IPC:Examples`, as follows:

- For the client, source code for a timing example, 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.

NuBus card - to - NuBus card

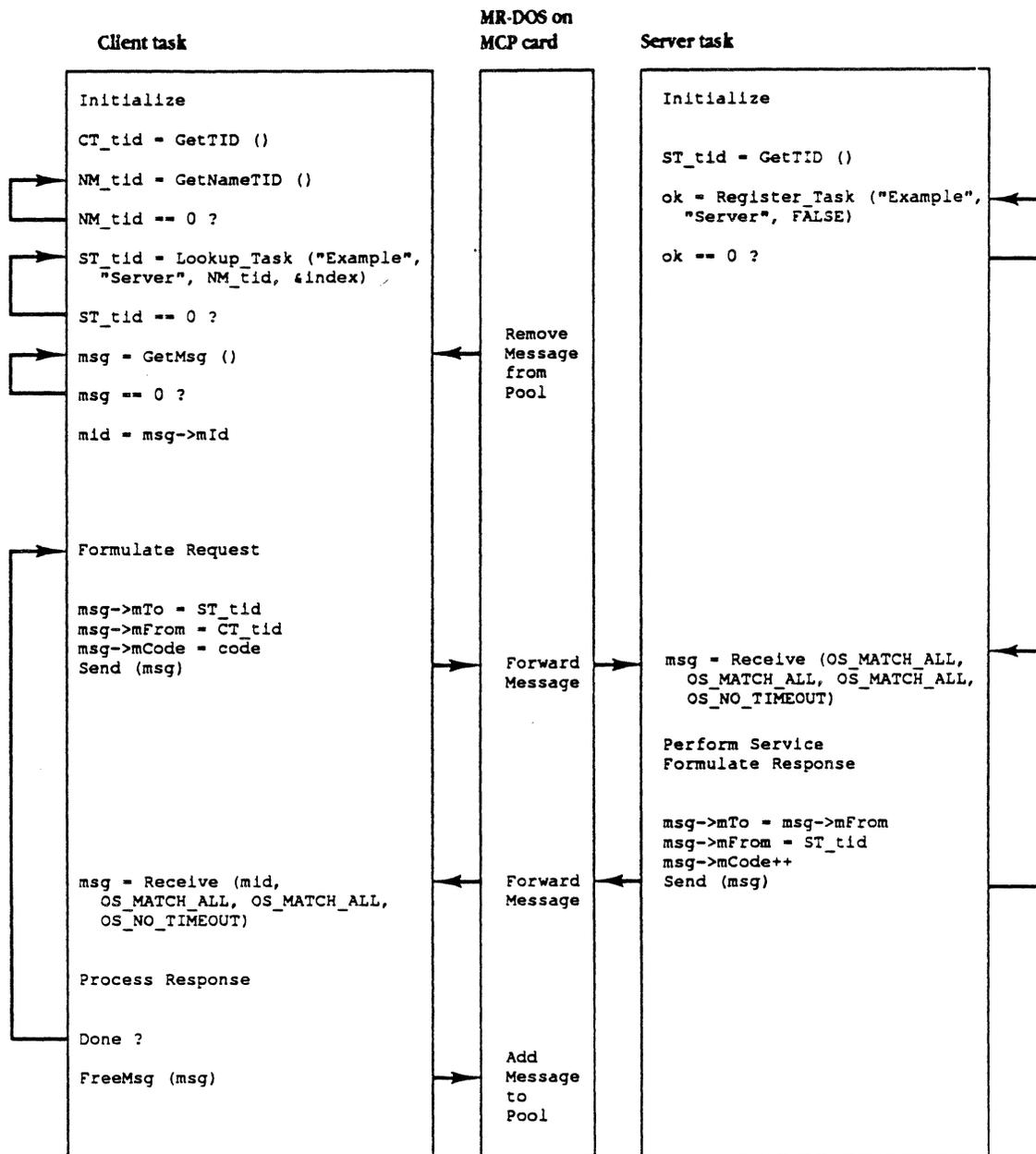


Fig. 3-5 (L7)
MCP Developer's Guide
Apple Computer, Inc.
JOYCE ZAVARRO
Illustrator 88
GEORGE M. VRANA

A server or client running under MR-DOS automatically has a TID associated with it; a server or client using the Apple IPC driver on the Macintosh II 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 MR-DOS so that clients can find it.

Under MR-DOS, both the server and the client can issue a blocking `Receive` request. MR-DOS has separate stacks for each task and saves each task's registers when switching between tasks. Using Apple IPC on the Macintosh II, only one process at a time (either the server or the client) can issue a blocking `Receive` request. Since the Macintosh II 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 Apple IPC driver on the Macintosh II, the `Receive` request has an extra parameter. This parameter is the address of a completion routine to be called when the Apple IPC 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.

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 Apple IPC 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.

The Apple IPC 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 careful in what it does in 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 Apple IPC driver that the client is finished talking to the IPC driver.

Figure 3-6 illustrates the process between the client/server relationship for applications using the Apple IPC 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 a completion routine so that the completion routine will be called when there is another message.

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

MSC NNNN
 ART: NN x 17 pi
 20.5 pi text to FN b/b

Figure 3-6
 The client/server relationship for applications using the Apple IPC driver

Using task scheduling in a multitasking environment

This section discusses the elements of 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, Trace Manager; and any developer-written tasks.

Task Identifiers

Tasks are known by and referred to MR-DOS by task identifiers. These identifiers are for internal use and are automatically assigned by MR-DOS when it starts a task.

Modes in which tasks run

There are two modes in which tasks run:

- Run-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 `Receive` call) or by using a MR-DOS 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.

Macintosh II to Macintosh II

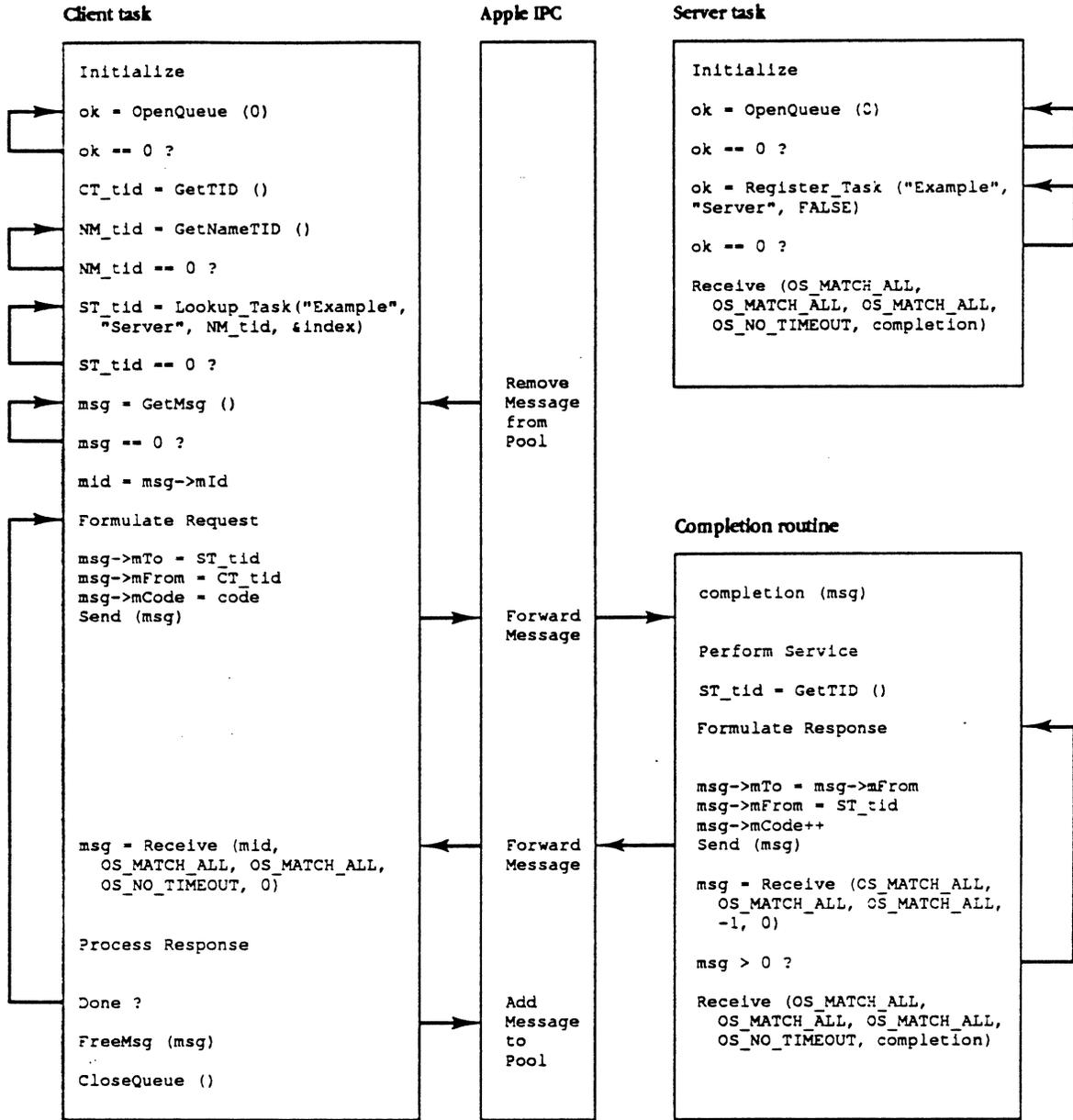


Fig. 3-6 (L8)
MCP Developer's Guide
Apple Computer, Inc.
JOYCE ZAVARRO
Illustrator 88
GEORGE M. VRANA

- ❖ *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 MR-DOS primitive `Reschedule()`.

Timer services

You can schedule tasks using timer services provided on the MCP distribution disk. 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.

Warning

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
- getting its own task identifier
- 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 processes 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 for 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.

MR-DOS always creates one task during its initialization; that task is the **Idle task**. The Idle task is one that 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, MR-DOS schedules the Idle task. Code segments can be run when MR-DOS is idle by installing them in the Idle Chain (refer to Chapter 7 for more information).

Warning

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 management

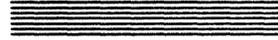
A distinction is made between using general-purpose memory and using message buffers to increase performance of the MR-DOS operating system.

For general-purpose memory, the available pool of memory is initialized as 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 is of a size you specify. Therefore, the amount of memory available in the pool depends on the sizes of the code and data space. You can allocate and free general-purpose memory to tasks using the `GetMem` and `FreeMem` calls, described in Chapter 4.

For message buffers, during initialization the operating system sets aside a block of memory large enough to hold a 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()` in `main()`.



Chapter 4



MR-DOS Primitives

This chapter describes the operating-system primitives used for MR-DOS. 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 mode.

Table 4-1 lists the primitives provided by MR-DOS operating system and gives a brief description of each.

Table 4-1
MR-DOS 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 ()	Reschedules a task for a later time
Send ()	Sends a message
Sp1 ()	Sets the hardware-priority level
StartTask ()	Initiates a task
StopTask ()	Stops a task

These primitives are calls that are made for a variety of processes in the MR-DOS. Some primitives are used in `main ()`, the program that executes before anything else starts. You can modify `main ()` for whatever application you are doing.

- ❖ *Note:* MR-DOS 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 types of calls take arguments and use similar data structures.

FreeMem()

`FreeMem()` frees a block of memory that was acquired earlier by a call to `GetMem()`. MR-DOS decrements the usage count associated with the buffer. If the resulting usage count is zero, the memory is returned to the free 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

In most cases, MR-DOS will execute an illegal instruction if an attempt is made to free a memory buffer that has not been allocated by MR-DOS.

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, MR-DOS 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 MR-DOS is freed using `FreeMsg()`.

GetMem()

`GetMem()` requests a block of memory from the free memory pool. The size of the free memory pool size 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 zero. A call to `FreeMem()` releases the memory. The returned 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 MR-DOS library routines `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 zero. A call to `FreeMsg()` releases the message.

MR-DOS clears all fields in the message, except Message ID and From address, before the pointer to the message is returned. Message ID (mID) is set to a number that is statistically unique to the field. From address (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`.

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, zero 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 zero.

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, MR-DOS 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 `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 4-2. 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 4-2
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_IMMEDIATE	Slice	No
OS_BLOCK_IMMEDIATE	Block	No
OS_RTN_MODE	Does not change	Yes
OS_RTN_IMMEDIATE	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_IMMEDIATE 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_IMMEDIATE 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_IMMEDIATE 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 *m;

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
*/
}
```

```

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, MR-DOS 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; MR-DOS 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 MR-DOS 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 MR-DOS, 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
```

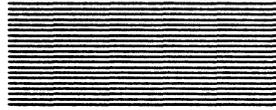
`P1` can be specified as a register (A0-A6, D0-D7) or as an immediate value (`#<abs-expr>`) or it can use any 68000 addressing mode valid in an LEA instruction to specify the location of a long word containing the desired task ID.

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.

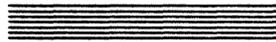
- ❖ *Note:* If a task calls `StopTask ()` and specifies its own task identifier, the task will kill itself and stop your program. To stop a task on a different smart card that is also running MR-DOS, send a message to the Remote System Manager on the other card.

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



MR-DOS Utilities

This chapter describes the operating system utilities available with MR-DOS. A utility is a library code segment linked with your application. Table 5-1 lists the MR-DOS utilities, and provides a brief description of each.

Table 5-1
MR-DOS utilities

Name	Description
BlockMove ()	Copies a block of data from the source address to the destination address on the same card
CopyNuBus ()	Copies a block of data from the source address to the destination address between on-card and off-card buffers
Date2Secs ()	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
GetDateTime ()	Returns the number of seconds between midnight, January 1, 1904, and the time that the function was called
GetETick ()	Returns the number of major ticks since the operating system 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 the buffer
IncUCount ()	Returns the incremented usage count of the buffer
IsLocal ()	Returns an indication of whether or not an address is local

MR-DOS utilities (continued)

Name	Description
Lookup_Task ()	Returns the task identifier of the task that matches the Object Name and the Type Name specified
MapNuBus ()	Translates a pointer that may contain a global address (NuBus address) to a local pointer
Register_Task ()	Registers a task with the Object Name and the Type Name specified
Secs2Date ()	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 pointer to a global address (NuBus address)
TraceReg ()	Registers the current task as the Trace Manager

A description of utilities

This section describes each of the operating system utilities and provides examples of the C declarations for each utility. This section also describes the assembler macros; these macros have a one-to-one relationship to the calls and require the same number of parameters. MR-DOS uses C calling conventions, and all registers are preserved except D0, D1, A0, and A1. MR-DOS 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 MR-DOS; the calling sequences and structures for these routines are defined in the file `os.h` in the folder `:MR-DOS:includes:`. These routines are identical to the routines `GetDateTime()`, `Date2Secs()`, and `Secs2Date()` 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`. The source and destination addresses must both be on the same card, otherwise use `CopyNuBus()`.

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;
    unsigned short 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
```

CopyNuBus()

`CopyNuBus()` copies a block of data from the source to the destination across the NuBus, without checking for overlapping source and destination addresses. The source address, destination address, or both may be main memory of the Macintosh II or memory on a smart card. The number of bytes is specified in `count`. The file `:Apple IPC:Examples:pr_manager.c` contains a sample program that uses `CopyNuBus`.

Caution

This routine deals with the complexity of potential 32-bit NuBus addresses for the source, the destination, or both, but does not deal with the possibility of overlapping buffers. Therefore, overlapping the source and destination blocks could cause partial overwriting of the destination block.

The C declaration for `CopyNuBus()` is

```
void CopyNuBus ( source, destination, count )
    char      *source;      /* Address of source buffer */
    char      *destination; /* Address of destination buffer */
    unsigned short count;  /* Byte count */
```

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

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

Date2Secs()

`Date2Secs()` takes the given date/time record, converts it to the corresponding number of seconds elapsed since midnight, January 1, 1904, and returns the result in the `secs` parameter.

The C declaration for `Date2Secs()` is

```
pascal void Date2Secs(Date, secs)
    DateTimeRec Date;
    long      *secs;
    extern;
```

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

```
#include    "os.h"
main()
{
    unsigned long secs;
    DateTimeRec 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
```

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 )
char      *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. For the slot number, get the value of location `gSlotNum` in the `gCommon` data area.

GetDateTime()

GetDateTime() returns the number of seconds between midnight, January 1, 1904, and the time that the function was called.

The C declaration for GetDateTime() is

```
pascal void GetDateTime(secs)
    long *secs;
extern;
```

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

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

Refer to the utility Date2Secs() 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

```
unsigned long GetETick();
```

The following example shows how to call GetETick in assembly language. To return the number of major ticks, get the value of location gMajorTick in the gCommon data area.

```
JSR     GetETick
```

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

GetgCommon()

`GetgCommon()` returns the address of the MR-DOS operating system data area, `gCommon`. Refer to the include files on your distribution disk for the structure of `gCommon`.

The C declaration for `GetgCommon()` is

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

The following example shows how to call `GetgCommon` in assembly language. To return the `gCommon` address, get the value of the constant `gCommon`.

```
JSR      GetgCommon
MOVE.L   D0 -> A0      /* A0 contains the beginning */
                          /* address of the gCommon data area*/
```

GetHeap()

`GetHeap()` returns the address of the heap area allocated to the task. If no heap area has been allocated, `GetHeap` returns zero. The heap size is specified on the call to `StartTask`.

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
```

Warning

`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 zero. The C declaration for GetICCTID () is

```
tid_type GetICCTID ();
```

The following example shows how to call GetICCTID in assembly language. To get the task identifier of the InterCard Communication Manager, get the value of 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

```
tid_type GetNameTID ();
```

The following example shows how to call GetNameTID in assembly language. To get the task identifier of the Name Manager, get the value of 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 zero. The C declaration for GetStParms () is

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

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

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

Warning

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

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

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

```
JSR      GetTCB
```

GetTickPS()

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

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

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

```
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` in assembly language. For the task identifier of the calling task, get the value of the location `gTID` in the `gCommon` data area.

```
JSR      GetTID
```

GetTimerTID()

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

The C declaration for GetTimerTID() is

```
tid_type GetTimerTID ();
```

The following example shows how to call GetTimerTID in assembly language. For the task identifier of the Timer Manager, get the value of the location gTimerTask in the gCommon data area.

```
JSR      GetTimerTID
```

GetTraceTID()

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

The C declaration for GetTraceTID() is

```
tid_type GetTraceTID ();
```

The following example shows how to call GetTraceTID in assembly language. For the task identifier of the Trace Manager, get the value of the location gTraceTask in the gCommon data area.

```
JSR      GetTraceTID
```

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 increases. A return value of 0 indicates that the pointer passed was 0.

The C declaration for GetUCount () is

```
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 the 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 orderly release of the buffers. IncUCount () 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 the 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 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 (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.

❖ *Note:* Lookup_Task () communicates with the Name Manager and issues a blocking Receive; therefore, the task gives up control of the CPU during this call.

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()` translates a pointer that may contain a global address (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.

❖ *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 may be active at any given time on each card; each call to `MapNuBus()` invalidates any mapping established by 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. The `MapNuBus` macro generates code in-line; 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
```

The file :MR-DOS:Examples:pr_manager.c contains a sample program that uses MapNuBus.

Caution

To move data across the NuBus, use CopyNuBus(). Tasks that use MapNuBus() must assume the responsibility for checking NuBus boundaries. Some hardware cards, including the MCP card, have a 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. CopyNuBus() checks for and correctly handles these boundaries.

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.

- ❖ *Note:* Register_Task() communicates with the Name Manager and issues a blocking Receive; therefore, the application gives up control of the CPU during this call.

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    D0                  ; check if register ok
BNE.S    OK                  ; jump if OK

```

Secs2Date()

`Secs2Date()` takes the number of seconds elapsed since 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 `Secs2Date()` is

```

pascal void Secs2Date(secs, Date)
    long      secs;
    DateTimeRec *Date;
extern;

```

The following example shows how to call `Secs2Date` from assembly language:

```

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

JSR      Secs2Date

```

Refer to the utility `Date2Secs()` 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. The ToNuBus macro generates code in-line; no registers are destroyed, except the specified registers. 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 46.

The C declaration for `TraceReg()` is

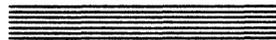
```
void TraceReg ();
```

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

```
JSR TraceReg
```



Chapter 6



MR-DOS Managers

This chapter describes the operating system managers provided with MR-DOS. A manager is a task that provides a set of services to other tasks; each manager is specific to a certain function.

MR-DOS Managers

Table 6-1 lists the managers provided with the MR-DOS operating system and a brief description of each.

Table 6-1
MR-DOS 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
❖ <i>Note:</i>	The Timer Manager is provided in this version of the MR-DOS software for historical purposes; it will not appear in the next version.
Trace Manager	Sends copies of all messages to a Trace Monitor (if available) for debugging purposes

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 for:

- 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, 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 Apple IPC 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 the size indicated in `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 MR-DOS 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 the 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 an Object Name. Each name is a maximum of 32 characters long. (For more detailed information on Type Names and Object Names, refer to *Inside AppleTalk*.)

- ❖ *Note:* Any character may be used; however, the equal sign (=), a wildcard character, should be avoided since it is not possible to match it explicitly.

The parameters in the message request to look up names, look up task identifiers, and register tasks are passed in a buffer associated with the message. 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 the names or task identifier using two 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, MR-DOS 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 MR-DOS 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 Canceling 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 Canceling 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 terminates. The Name Manager immediately replies to the request, indicating whether the remote task is currently running or not. The remote task is considered to be terminating 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 using two Name Manager messages:

- NM_REG_TASK
- NM_UNREG_TASK

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 requests 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

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 6-4
Printf standard conversion

Character	Standard conversion
<code>%d</code>	decimal conversion
<code>%u</code>	unsigned conversion
<code>%x</code>	hexadecimal conversion
<code>%X</code>	hexadecimal conversion with capital letters
<code>%o</code>	octal conversion
<code>%c</code>	character
<code>%s</code>	string
<code>%m.n</code>	field width, precision
<code>%-m.n</code>	left adjustment
<code>%0m.n</code>	zero-padding
<code>%.*. *</code>	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 lists the nonstandard conversion characters that `printf` also supports.

Table 6-5
Printf nonstandard conversion

Character	Non-standard 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.

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 global (NuBus) address of the memory to be released.

The calling parameters for `RSM_FreeMem` are as follows:

```
mCode      RSM_FreeMem
mDataPtr   Global (NuBus) 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, MR-DOS on the remote card executes an illegal instruction if an attempt is made to free a memory buffer that has not been allocated by MR-DOS.

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
mOData[0]  Address of buffer (as returned to RSM), or
           0 if not allocated
mDataPtr   Global (NuBus) address of the buffer, or
           0 if not allocated
mStatus    RSE_NO_ERRORS
```

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
```

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.

Finding the Remote System Manager

Tasks can determine the task identifier of a Remote System Manager on another card by following 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 using the Remote System Manager on the remote card. To do so, refer to the file `Apple IPC:Examples:RSM_tester.c` for annotated code.

- ❖ *Note:* If errors occur, then any allocated memory must be returned 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 zero. If zero is returned or if errors are detected, then any allocated memory must be returned.

Caution

When this example requests memory for the code to be downloaded to the card, it does not determine the size of this code; this example has a hard-coded number. If you develop your code using this `RSM_tester` example, be sure to change the amount of hard-coded memory that was allocated on the card for the task to be downloaded.

Timer library and Timer Manager

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

Warning

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. `TLStartTimer` 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.

TActiveTimer()

The `TActiveTimer()` call returns a count of the number of active timer messages.

```
long TActiveTimer (topb, mID)
TOPB *topb;
long mID;
```

If `mID` is not zero, `TActiveTimer()` returns 1; if the message corresponding to the `mID` is active, `TActiveTimer()` returns 0; if the `mID` is zero, `TActiveTimer()` 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.

Warning

This section describing the Timer Manager is included in this document for historical purposes only. It is strongly recommended that you use the timer library. In future releases, the Timer Manager will be removed.

Table 6-6 lists the Timer Manager calls and functions.

Table 6-6
Timer Manager calls

Function	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 Apple IPC 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 contents of size `mDataSize`. The syntax of the `TraceMonitor` 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 MR-DOS using a `TraceReg` call and begins tracing. The MR-DOS `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 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:

<code>mCode</code>	<code>TM_TRACE</code>
<code>mDataPtr</code>	Pointer to copy of traced message (and data)
<code>mDataSize</code>	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:

- `-an` dump messages having To or From values of *n*
- `-cn` dump messages having Code value of *n*
- `-dn` dump messages having DataPtr value of *n*
- `-fn` dump messages having From value of *n*
- `-in` dump messages having ID value of *n*
- `-ln` dump messages having DataSize value of *n*
- `-pn` dump messages having Priority value of *n*
- `-sn` dump messages having Status value of *n*
- `-tn` 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.

Example 1:

```
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).

Example 2:

```
DumpTrace -a0d000003
```

In this example, DumpTrace dumps from TraceFile those messages that are either to or from slot d, task 3.

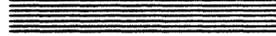
The following example of DumpTrace shows output for a message with an associated data buffer:

```
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 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 MR-DOS

This chapter describes methods to handle peculiarities of MR-DOS, 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 `CopyNuBus` utility routine copies data from a source to a destination and handles off-card buffers. Following is an example that demonstrates a simple case of copying a buffer from one place to another using `CopyNuBus`.

```
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 >>
}
```

Caution

It is recommended that you use `CopyNuBus()` rather than `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, `CopyNuBus()` checks for and correctly handles the boundaries.

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`.

Inter-card buffer copy

Any piece of code that manipulates more than one potentially off-card buffer at a time can be complex. For example, copying between two such buffers results in the operating system continually calling `MapNuBus` to adjust the mapping hardware. This operation may actually be more efficient if the data is copied through an intermediate local buffer.

```
CopyNubus (mPtr1->mDataPtr, mPtr2->mDataPtr, mPtr1->mDataSize);
```

Inter-card message passing

Normally, there is no need to be concerned about how messages are moved from one card to another, since MR-DOS handles these transparently to the user 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. ICCM adjusts the `mDataPtr` field of the message, if necessary, to ensure that the pointer is valid across the NuBus.

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.

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 MR-DOS 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`, MR-DOS 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 `MR-DOS:Examples:pr_manager.c` and `MR-DOS:Examples:osscint.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
      MapAddr  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 MR-DOS 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 MR-DOS global variables.

- ❖ *Note:* Any routine not loaded with the MR-DOS 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, MR-DOS 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  $\pi$  shop rental calculator */
    idlenextcall ();
}

```

Writing your own download program

If you want to dynamically download tasks to an MCP-based card, you can create your own download program. MCP provides two subroutines, Findcard and Download, to help you write your own download program.

Two binary versions of each of the subroutines are provided on the MCP distribution disk:

- one version for the MCP card, found in the :MR-DOS:MCP:Download-lib.o library
- another version for the AST-ICP card, found in the :MR-DOS:AST_ICP:Download-lib.o library.

Findcard subroutine

The Findcard subroutine allows your program to locate smart cards in the Macintosh II, using the following code:

```
pascal short Findcard(slot, type)
short      *slot;
long       type;
extern;
```

where: **slot* contains the address of a bit mask indicating which slots are available for loading

type is the type of card to download

- ❖ *Note* The *type* field is designed for use at some future date; currently, *type* is unused and should be zero.

In the bit mask, bit 0 is the least significant bit. Bit 9 corresponds to slot 9, and bit 14 corresponds to slot E. Findcard returns `DLE_NOERR` (no error) if a card of the correct type is found, or returns `DLE_EMPTY` if no cards of the correct type are found.

Download subroutine

The Download subroutine allows you to download a specified file, appropriate address, slot, and registers into the MR-DOS operating system, using the following parameters:

```
pascal short Download(FileName, vRef, slot, loadaddr, initial_load,
registers, type)

char      *FileName;           /* The file name is a C string. */
short     vRef;
short     *slot;
long      loadaddr;
short     initial_load;
struct    ST_Registers        *registers;
long      type;
extern;
```

The **FileName* field is a pointer to the filename of a file to load. The filename is a C string.

The *vRef* field is the volume reference number of a file to load.

The **slot* field is a bit mask indicating which slots to load. Bit 0 is the least significant bit. Bit 9 corresponds to slot 9. Bit 14 corresponds to slot E. If the bit mask equals `0XFFFF`, all cards that are of the correct type will be downloaded.

The `loadaddr` field is the relative address on a smart card to load data and/or code. The default initial load address of MR-DOS is defined by the symbol `INIT_LOAD` in the file `siop.h` in the `:MR-DOS:includes:` folder.

The `initial_load` field is 0 if a non-initial load, non-zero if an initial load. If it is an initial load, the card is reset, memory is cleared, and the card is restarted with the PC from the newly loaded data and/or code for a non-initial load to work. MR-DOS must already be running on the card. The calling task must have previously obtained the memory on the card where the code is to be loaded.

❖ *Note:* It is the responsibility of the calling task to issue a `StartTask` request if the calling task specified a non-initial load; MR-DOS returns the registers to be passed to `StartTask`. It is the responsibility of the calling task to set up all other parameters to `StartTask`. For an example of an MPW tool that will do a non-initial load, see the file `:Apple IPC:Examples:RSM_tester` on the MCP distribution disk.

The `ST_Registers` field is a pointer to a register area (defined in the file `os.h` in the `MR-DOS:includes:` folder) where the correct registers are returned for use in a `StartTask` request. The Program Counter (PC) and Stack Pointer (SP) used by the Download subroutine are returned to this area on an initial load.

The `type` field is the type of card to download. This feature is designed for use at some future date; currently, `type` is ignored and should be zero.

Download errors

Download errors are indicated by messages to the `stderr` file. The state of any card to be downloaded is undefined if an error is returned. `DLE_NOERR` is a normal return. Table 7-1 lists Download error constants; these constants are found in the `include` file in the folder `:MR-DOS:includes:Download.h`.

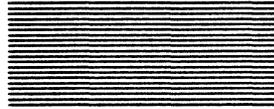
Table 7-1
Error constants for Download

Error Displayed	Number	Description
DLE_NOERR	0	No error
DLE_NOJT	1	No jump table found
DLE_DATAINIT	2	Bad Data Init segment
DLE_GLOBALF	3	Global-data format error
DLE_CODES	4	Code segment error
DLE_MAC2	5	Code only runs on Macintosh II
DLE_EMPTY	6	No cards found
DLE_NOCARD	7	Slot specified is empty
DLE_RESFILE	8	Couldn't open resource file
DLE_FILEWRONG	9	Download file is wrong type

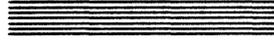
The include file `:MR-DOS:includes:Download.h` contains the following Download constant that may be useful for development:

```
#define Max_Slots 16 /* Max number of card slots */
```

You can also find the routine declarations for Download and Findcard in this include file.



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 using MCP
- how to get code running on the MCP card
- where to start debugging the program

What you will develop

You will 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 will create the following:

- a program module containing a MR-DOS task that will be downloaded to the smart card
- an application program to run on the Macintosh II main logic board that incorporates Apple IPC, the driver that interacts with the MR-DOS task on the smart card program

Before you start

Before learning how to create these 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 a program module to be downloaded to the MCP card:

- MR-DOS:Examples:osmain.c
- MR-DOS: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 MR-DOS and Apple IPC.

The examples in this chapter demonstrate how to build a MR-DOS program module and download it to an MCP card. Development is similar for building an application program using Apple IPC 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 MR-DOS as though it were a normal Macintosh application.

You should avoid normal Macintosh run-time libraries; the Macintosh toolbox is not supported by MR-DOS.

How to create applications using MCP

In order use MCP 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
- ❖ *Note:* 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 MR-DOS library routines

Makefiles are supplied as examples to illustrate the creation of applications for both MR-DOS and Apple IPC. 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).

Create 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 - MR-DOS          */
/*                                          */
/*****

#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);
    }
}

/*****

```

Modify the main program

The main program initiates both the tasks and MCP software (including MR-DOS and supporting software services).

The file `:MR-DOS: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 MR-DOS, define and start a number of tasks, set the clock rate, and then pass control to MR-DOS.

The main program you create should consist of:

- a call to `osinit()` to initialize MR-DOS
- ❖ *Note:* 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 MR-DOS 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.

You should have already created a new main program file by copying the `osmain` file from the folder `:MR-DOS: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 - MR-DOS
/*
/*
/*      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
#ifdef MCP
    cp_ptr = NULL;
#endif

    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 - MR-DOS
/*
/*
/*          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  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 `:MR-DOS:Examples`, and are now ready to modify that new file. Using the makefile, you can:

- compile the initialization software (`osmain.c`) and application tasks
- link the desired MR-DOS 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 MR-DOS operating system does not support the Macintosh toolbox.

MR-DOS include files

Table 8-1 lists the include files available and briefly describes each file. These include files are located in the folder `:MR-DOS:includes:` on the MCP distribution disks. You will also use these files to compile and link your code using the makefile.

Table 8-1
Include files

Assembly Filename	C Language Filename	Description of File
<code>os.a</code>	<code>os.h</code>	Defines the operating-system message structure, commonly used constants, and externally visible system library routines.
<code>managers.a</code>	<code>managers.h</code>	Contains the structures and constants used when accessing the Name Manager, Time Manager, and InterCard Communications Manager.
<code>mrDOS.a</code>	<code>mrDOS.h</code>	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 `MR-DOS:includes:` folder on the distribution disks. These files are useful if any SCC hardware is to be used.

MR-DOS libraries

The file :MR-DOS:MCP:os.o is the library containing MR-DOS operating system routines for the MCP card. The file :MR-DOS:MCP:osglue.o is the glue (interface) library containing code to allow tasks to use MR-DOS utility routines. (Equivalent files for the AST-ICP card are :MR-DOS:AST_ICP:os.o and :MR-DOS:AST_ICP:osglue.o.)

- ❖ *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.

You must link your code with these files using the MPW Link command.

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 -d Card=MCP           ...   for generating code for the MCP card or
#           make -d Card=AST_ICP       ...   for generating code for the ast card

```

```

CI                =      ::includes:
LinkOpts         =      -l -x :"{Card}":xref > :"{Card}":map

AOptions        =      -d &Card=ð'{Card}ð' -i ::includes:,:"{Card}": -l -font Courier,ð
7 -pagesize 115,124 -print Data,Obj,Lits,NoMDir
COptions        =      -D"{Card}" -DPRINT -g -i {CI}
CSources        =      ::includes:scc.h echo.c trace_manager.c pr_manager.c ð
printf.c name_tester.c timer_tester.c osmain.c timeIt.c ð
l3osmain.c L3MMSVP.c L3MMSVPClient.c
AsmLists        =      ossccint.a.lst IOPNub.a.lst L3MMSVP.a.lst
Targets         =      =.o =.lst start

: "{Card}":      f      : ::includes:

all              f      : "{Card}":start

: "{Card}":start f      : "{Card}":osmain.c.o : "{Card}":ossccint.a.o ð
: "{Card}":OS.o : "{Card}":osglue.o : {Card}:pr_manager.c.o ð
: "{Card}":printf.c.o : "{Card}":name_tester.c.o ð
: "{Card}":timer_tester.c.o : "{Card}":L3MMSVP.c.o ð
: "{Card}":L3MMSVPClient.c.o : "{Card}":L3MMSVP.a.o ð
: "{Card}":timeIt.c.o : "{Card}":echo.c.o ð
: "{Card}":trace_manager.c.o

If "{Card}" == "MCP"
Link -t 'GMSC' -c 'RWM' -o : "{Card}":start ð
: "{Card}":osmain.c.o : "{Card}":ossccint.a.o : "{Card}":OS.o ð
: "{Card}":osglue.o : "{Card}":pr_manager.c.o : "{Card}":printf.c.o ð
: "{Card}":name_tester.c.o : "{Card}":timer_tester.c.o ð
: "{Card}":timeIt.c.o : "{Card}":echo.c.o ð
: "{Card}":trace_manager.c.o : "{Card}":L3MMSVP.c.o ð
: "{Card}":L3MMSVPClient.c.o : "{Card}":L3MMSVP.a.o {LinkOpts}

Else
Link -t 'GMSC' -c 'RWM' -o : "{Card}":start ð
: "{Card}":osmain.c.o : "{Card}":ossccint.a.o : "{Card}":OS.o ð
: "{Card}":osglue.o : "{Card}":pr_manager.c.o ð
: "{Card}":printf.c.o : "{Card}":name_tester.c.o ð
: "{Card}":timer_tester.c.o : "{Card}":timeIt.c.o ð
: "{Card}":echo.c.o : "{Card}":trace_manager.c.o ð
: "{Card}":L3MMSVP.c.o : "{Card}":L3MMSVPClient.c.o ð
: "{Card}":L3MMSVP.a.o {LinkOpts}

End

: "{Card}":osmain.c.of {CI}os.h {CI}managers.h {CI}mrDOS.h {CI}siop.h
: "{Card}":ossccint.a.o f : "{Card}":OSDefs.d
: "{Card}":printf.c.o f {CI}os.h {CI}managers.h
: "{Card}":echo.c.o f {CI}os.h
: "{Card}":trace_manager.c.o f {CI}scc.h {CI}siop.h {CI}os.h {CI}managers.h
: "{Card}":name_tester.c.o f "{CI}"os.h "{CI}managers.h {CI}mrDOS.h

```

```

: "{Card}":timer_tester.c.o f      "{CI}"os.h  "{CI}managers.h
: "{Card}":timeIt.c.o      f      "{CI}"os.h  "{CI}managers.h
: "{Card}":L3MMSVP.c.o    f      "{CI}"os.h  {CI}diags.h
: "{Card}":L3MMSVP.a.o f      "{CI}"MRDOS.a  "{CI}os.a  {CI}diags.a  {CI}slop.a
: "{Card}":L3MMSVPClient.c.o f      "{CI}"os.h  {CI}  diags.h  "{CI}managers.h

# Special targets.

# Listings - Print changed files.

Listings ff      {AsmLists}
Print -f Courier -s 7 -ls 0.70 -r {NewerDepts}

Listings ff      {CSources}
Print -f Courier -s 7 -ls 0.70 -r -hf Courier -hs 9 -h -n {NewerDepts}
echo "Last listings made `Date`." > Listings

# Clean - Remove all targets.

Clean f      {Targets}
Delete -i {Targets}
/*****/

```

The resulting makefile should look as follows:

```

/*****/
/*
/*          Edited Makefile for download example.          */
/*
/*
/*          Copyright © 1987, 1988 Apple Computer, Inc. All rights reserved. */
/*
/*****/

# Since we are building a program for the MCP smart card, hardcode
# Card to be 'MCP'

Card      =      MCP
# Points to the new MCP folder on your hard disk.

MRDOS     =      ::MCP Software:MR-DOS:
CI        =      {MRDOS}includes:
LIBDIR    =      {MRDOS}MCP:
LinkOpts  =      -l -x xref > map

AOptions  =      -d &Card=@'{Card}@' -i "{CI}","{LIBDIR}" -l -font Courier,7 @
            -pagesize 115,124 -print Data,Obj,Lits,NoMDir
COptions  =      -D"{Card}" -DPRINT -g -i "{CI}"
ExampleBins =      {MRDOS}Examples:{Card}:

```

```

CSources      =      osmain.c NewTask.c
AsmLists      =
Targets       =      *.o *.lst start

:      f      :      "{MRDOS}"includes: "{ExampleBins}"

all           f      start

start f      osmain.c.o "{LIBDIR}"OS.o "{LIBDIR}"osglue.o @
              "{ExampleBins}"printf.c.o "{ExampleBins}"trace_manager.c.o @
              NewTask.c.o
link -t 'GMSC' -c '????' -o start @
              osmain.c.o "{LIBDIR}"OS.o @
              "{LIBDIR}"osglue.o @
              "{ExampleBins}"printf.c.o @
              "{ExampleBins}"trace_manager.c.o @
              NewTask.c.o {LinkOpts}

osmain.c.o f      "{CI}"os.h  "{CI}"managers.h  "{C}"mrdos.h  "{CI}"siop.h
NewTask.c.o f      "{CI}"os.h

"{ExampleBins}"printf.c.o f      "{MRDOS}(:Examples:printf.c  "{CI}"os.h
"{CI}"managers.h

"{ExampleBins}"trace_manager.c.o f      "{MRDOS}(:Examples:trace.manager.c @
                                      "{CI}"scc.h "{CI}"siop.h "{CI}"os.h @
                                      "{CI}"managers.h

#      Special targets.

#      Listings - Print changed files.

Listings      ff      {AsmLists}
              Print -f Courier -s 7 -ls 0.70 -r {NewerDeps}

Listings      ff      {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 f      {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 "MCP" -D PRINT -i "::MCP Software:MR-DOS:"includes: osmain.c -o osmain.c.o
C -D "MCP" -D PRINT -i "::MCP Software:MR-DOS:"includes: NewTask.c @
  -o newTask.c.o
Link -t 'GMSC' -c '????' -o start @
  osmain.c.o "::MCP Software:MR-DOS:MCP:"OS.o @
  "::MCP Software:MR-DOS:MCP:"osglue.o @
  "::MCP Software:MR-DOS:Examples:MCP:"printf.c.o @
  "::MCP Software:MR-DOS:Examples:MCP:"trace_manager.c.o @
  NewTask.c.o -l -x xref > map
```

◆ *Note* {MR-DOS} is the pathname of the MR-DOS folder under MPW. You must set this up when using MPW; otherwise, you must substitute the full pathname for {MR-DOS}.

Table 8-2 defines the parameters to the `Link` command, shown in the example above.

Table 8-2
Link command parameters

Parameter	Description
<code>-t</code>	The type of file that <code>Link</code> command is going to generate
<code>GMSC</code>	The file type that the Downloader application looks for If you use the AST-ICP card during development instead of the MCP card, you must replace <code>GMSC</code> with <code>KARD</code> .
<code>-c</code>	The creator
<code>'????'</code>	Enter any appropriate creator name
<code>-o start</code>	The output file from the linker; the file <code>start</code> will be created in your directory
<code>osmain.c.o</code>	The initialization routine that you modified
<code>os.o</code>	File that contains MR-DOS operating system

Table 8-2 (continued)

Parameter	Description
<code>osglue.o</code>	File containing glue code
<code>printf.c.o</code>	Printing subroutine source code for MR-DOS; equivalent to the <code>printf</code> routine in standard C
<code>trace_manager.c.o</code>	Tracing tool for MR-DOS
<code>NewTask.c.o</code>	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. For development efforts, a version of Download is provided on the distribution disk for the MCP card and for the AST-ICP smart card. The makefile in `:MR-DOS:Examples` produces two executable files for downloading; these files are:

- `:MR-DOS:Examples:MCP:start`, code to be downloaded to the MCP card
- `:MR-DOS:Examples:AST_ICP:start`, code to be downloaded to the AST-ICP card

The file produced depends upon the `-d` option used with the make command in `:MR-DOS:Examples:.` This section first discusses the Download tool, then presents information to help you create your own download application.

Calling the Downloader 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 will 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:mr-dos:Examples:mcp:Download" start
```

Next, enter the following command:

```
directory "::MCP Software:Apple IPC: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 folder `:MR-DOS:includes:Download.h`.

Table 8-3
Error constants for Download

Error Displayed	Number	Description
<code>DLE_NOERR</code>	0	No error
<code>DLE_NOJT</code>	1	No jump table found
<code>DLE_DATAINIT</code>	2	Bad Data Init segment
<code>DLE_GLOBALF</code>	3	Global data-format error
<code>DLE_CODES</code>	4	Code segment error
<code>DLE_MAC2</code>	5	Code only runs on Macintosh II
<code>DLE_EMPTY</code>	6	No cards found
<code>DLE_NOCARD</code>	7	Slot specified is empty
<code>DLE_RESFILE</code>	8	Couldn't open resource file
<code>DLE_FILEWRONG</code>	9	Download file is wrong type

Debugging your code

You can use any debugger for the Macintosh II to examine or change data or code in your application. For this example, a subset of Macsbug is used to debug the application in memory on the MCP card.

The high-order nibble in the address specifies the card that is to be examined. For example, if you want to dump memory from the MCP card installed in slot B at location 400, type:

```
dm b00400
```

After detecting an illegal condition (via the exception 68000 vectors or hardware interrupt), a MR-DOS handler dumps the current register set to an area of card memory. This area of memory starts at 0x0600 on the MCP card where the exception/interrupt occurred. Table 8-4 lists the format of the dump area.

Table 8-4
Dump 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 StackPointer

SR is the Status Register at the time of the error

PC is the Program Counter at the time of the error

USP is the User Stack Pointer at the time of the error

Flag is a byte that starts at address 0x064A. It contains the value 0xFF when an error has occurred.

trap number is the 68000 exception ID

Examine the Flag byte at 0x064A. If it contains 0xFF, the system has crashed. Refer to Chapter 11 for more complete information on how to track system crashes and hangs.

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. Clearing this byte causes the registers to be reloaded with the saved registers and the system to be restarted.

❖ *Note:* This format is accurate only if `IOPNub.a` is not being used. `IOPNub.a` also dumps registers to area `0x0600`, but in a different format.

You can also force your code to crash by using the `ILLEGAL` assembler operator. This defines the C function `illegal()`, which when called generates an illegal instruction.

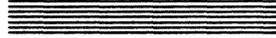
```
pascal void illegal ()
extern 0x4afc;
```

Next, enter the following sequence at the location chosen in your code:

```
if (condition == want to crash)
    illegal ();
```



Chapter 9



Apple IPC

As described in Chapter 3, Apple IPC provides services to Macintosh II programs or processes that are used to communicate with other processes on the Macintosh II or on one or more smart cards. Apple IPC includes MR-DOS message passing, task naming, and echo services; it is *not* another operating system for the Macintosh computer.

This chapter describes where to find Apple IPC on the MCP distribution disks, how to install and use Apple IPC, and how to make specific calls to Apple IPC.

The Apple IPC software

Apple IPC software consists of the Apple IPC driver, development tools, include files, and examples. The MCP distribution disks contains a folder named :Apple IPC: that contains the following:

- a file named `IPCGLue.o` that contains the Apple IPC library, providing object routines (glue code) for interfacing to the Apple IPC 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 Apple IPC, which contains
 - the Apple IPC 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 Apple IPC file that contains everything just described for the Apple IPC file, plus the Echo example.
- ❖ *Note:* The Echo example is almost identical to the Echo Manager, and is provided to show how you can add a manager to the Apple IPC file.
 - 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 Apple IPC driver

Each of these components is described in this chapter in the section on Apple IPC services, along with examples of C and assembly-language macros for each Apple IPC call.

Installing Apple IPC

As described in Chapter 2, here are the steps that you should follow to install Apple IPC into your Macintosh II.

❖ *Note:* It is not necessary to repeat these steps if you have already followed the instructions in Chapter 2.

1. **Open the ':Apple IPC': folder in the new MCP Software folder you created on the Macintosh II desktop.**
2. **Open the ':Examples:' folder and select the 'Apple IPC' file.**
3. **Copy the 'Apple IPC' file into the System Folder of the Macintosh II.**

❖ *Note:* To make this example as easy as possible, you can copy the file in one step as follows: hold down the Option key while dragging the 'Apple IPC' file into the System Folder.

4. **Reboot the Macintosh II.**

The Apple IPC driver is loaded into the system heap during system start-up by an INIT31 resource within the Apple IPC file.

Using Apple IPC

An application that uses Apple IPC must make an initial call to `OpenQueue` to establish its use of IPC. Each process that uses Apple IPC requests that a queue be opened by calling `OpenQueue`.

Messages are sent and received through Apple IPC using `Send` and `Receive`.

- When the Apple IPC 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.

Apple IPC services

This section describes the Apple IPC 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 MR-DOS for applications running on the Macintosh II computer.

- ❖ *Note:* As with MR-DOS, Apple IPC 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 Apple IPC, with a brief description of each.

Table 9-1
Apple IPC services

Name	Description
CloseQueue ()	Closes an Apple IPC queue
CopyNuBus ()	Copies a block of data from the source address to the destination address
FreeMsg ()	Frees a message buffer
GetCard ()	Returns the NuBus slot number on which the calling process is running
GetICCTID ()	Returns the task identifier of the InterCard Communication Manager
GetIPCg ()	Returns the address of the global data area within the Apple IPC 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 Apple IPC queue
Receive ()	Receives a message

Table 9-1 (continued)

Name	Description
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

CloseQueue()

CloseQueue () closes a queue that was previously opened. This IPC 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
```

CopyNuBus()

CopyNuBus () copies a block of data from the source to the destination, without checking for overlapping source and destination addresses. The number of bytes is specified in count. The source address and/or destination address may be Macintosh main memory or memory on a smart card.

Caution

This routine deals with the complexity of potential 32-bit NuBus addresses for the source and/or the destination, but does not deal with the possibility of overlapping buffers. Therefore, do not overlap the source and destination blocks, because doing so could cause partial overwriting of the destination block.

The C declaration for CopyNuBus () is

```
void CopyNuBus ( source, destination, count )
char *source; /* Address of source buffer */
char *destination; /* Address of destination buffer */
unsigned short count; /* Byte count */
```

The following example shows how to call CopyNuBus using assembly language.

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

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 Apple IPC resource entries of type aipn found in the Apple IPC 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.

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 Apple IPC driver that is configured into the System File.

GetIPCg()

GetIPCg() returns the address of the data area of the Apple IPC 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 Apple IPC 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.

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 (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), 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
```

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 IPC 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 Apple IPC; that is, it is a way of letting you check for operator termination.

This function must be called before any other call to IPC can be made. You can issue either

- an AppleIPC `CloseQueue` request, or
- a `KillReceive` request

If the procedure issues an AppleIPC `CloseQueue` request and returns to the Apple IPC 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 IPC `Receive` performs in exactly the same way as the MR-DOS `Receive` primitive.

The fifth parameter is the address of a C completion routine. This parameter is required for Apple IPC, 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.

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 multitasking operating system; therefore, the application cannot cease to process events. Under MR-DOS, a process can do a blocking `Receive` and permit other processes to execute.

Table 9-2 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 9-2
State table for the `Receive` call

Time-out value	Completion routine	Message available	Immediate results	Subsequent results
<0	No (0)	No	Returns 0 to the <code>Receive</code> request	None
	No (0)	Yes	Returns message to <code>Receive</code> request	None
	Yes	No	The Apple IPC driver returns 0 to the <code>Receive</code> request; the completion routine is not called	None
	Yes	Yes	The Apple IPC driver calls the completion routine with the message, after which the driver returns 0 to the <code>Receive</code> request	None
=0	No (0)	No	Waits until it gets a message, then returns a message to the <code>Receive</code> request	Waits for a message; <code>OpenQueue</code> routine is called continuously.
	No (0)	Yes	When a message arrives, returns a message to the <code>Receive</code> request	None
	Yes	No	Returns 0 to the <code>Receive</code> 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 <code>Receive</code> request	None

Table 9-2 (continued)

Time-out value	Completion routine	Message available	Immediate results	Subsequent results
> 0	No (0)	No	Waits for a message does not arrive If the time interval that you specify expires, then it returns 0 to the <code>Receive</code> request	<code>OpenQueue</code> routine is called continuously
	No (0)	Yes	Message returns to the <code>Receive</code> request	None
	Yes	No	Immediately returns 0 to the <code>Receive</code> 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 <code>Receive</code> 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 a 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.

Results returned

Whenever you call the `Receive` request on Apple IPC, you get one of three results returned from the IPC driver:

- 0
- message
- negative number (indicating an error)

Table 9-3 lists the two errors only that can be returned when a `Receive` request is made to Apple IPC.

Table 9-3
Errors returned

Error	Number	Description
<code>NoQueueErr</code>	-64	Error code for no more queues or bad queue
<code>QueueBusy</code>	-65	If <code>Receive</code> 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.

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.

Warning

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.

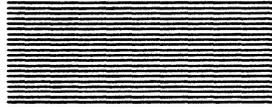
The C declaration of SwapTID() is

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

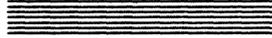
The assembly-language form for the SwapTID macro is as follows, where P1 is the address of the message buffer:

```
[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.



Chapter 10



Using the Forwarder with
Apple IPC

This chapter describes the Forwarder, tells how the Forwarder sends messages in conjunction with Apple IPC, 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 MR-DOS 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 MR-DOS 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.

MSC NNNN
ART: NN x 17 pi
20.5 pi text to FN b/b

Figure 10-1
Message paths using the Forwarder

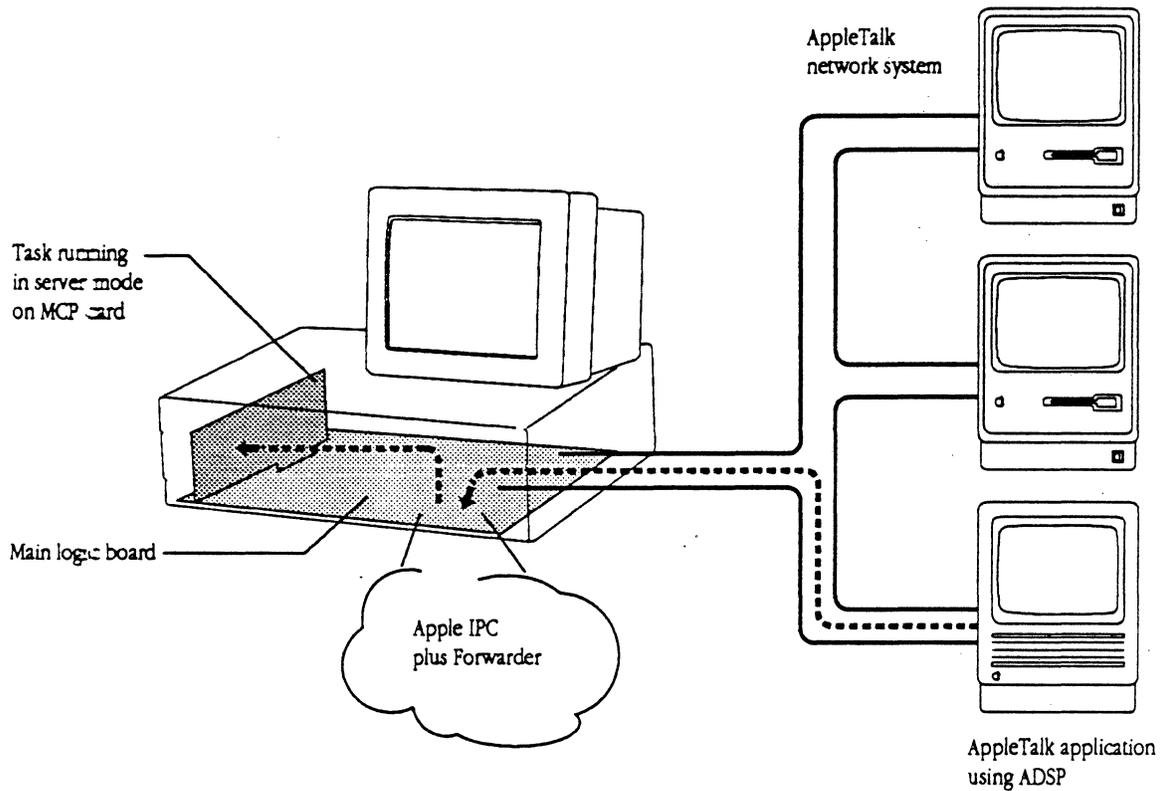


Fig.10-1 -COMP (L10)
MCP Developer's Guide
Apple Computer, Inc.
JOYCE ZAVARRO
Illustrator 88
GEORGE M. VRANA

How the Forwarder sends messages

The Forwarder sends messages when:

- a task running under MR-DOS 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 MR-DOS

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:

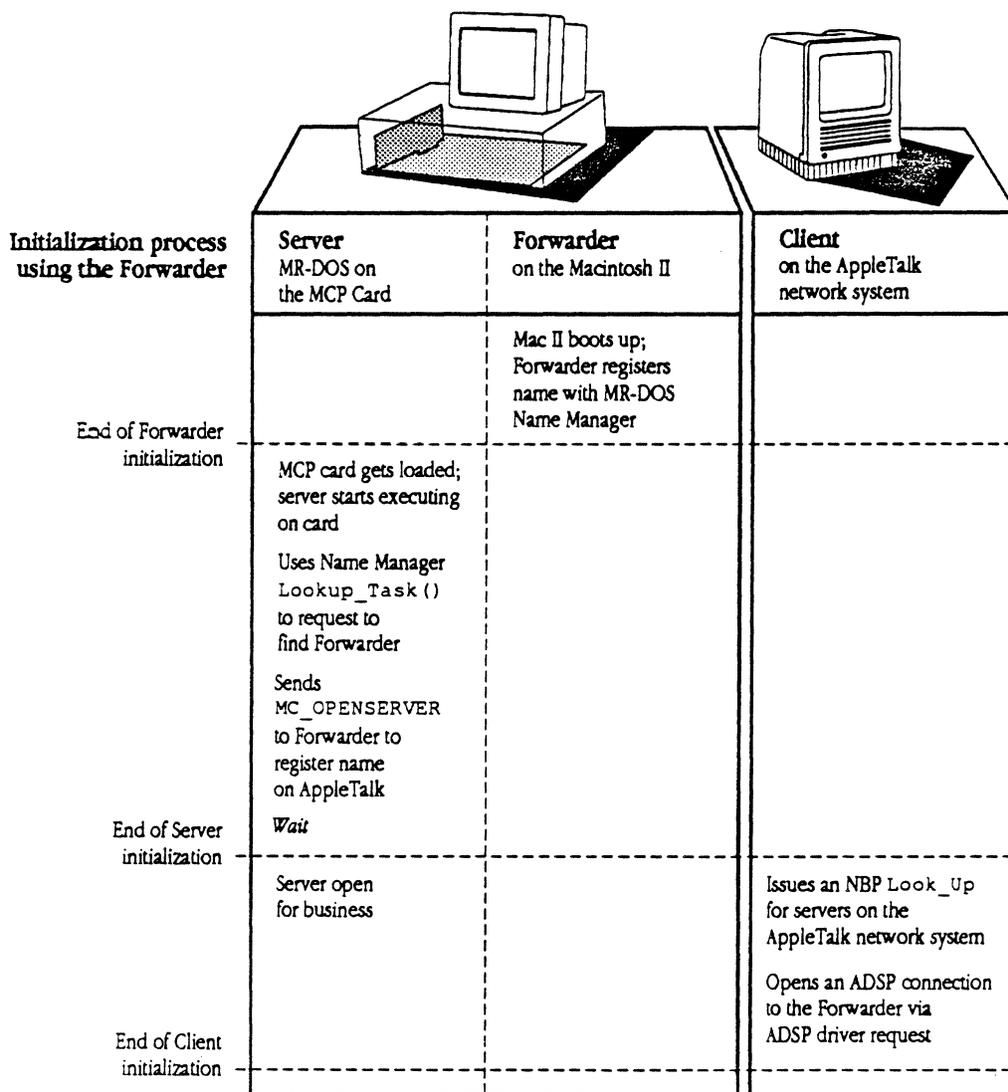
- `svcn`, which tells the Forwarder how much memory to preallocate for the server and for communications. The Forwarder will attempt to call for this number of free services and free (validate?) communication memory available.
- `sysz`, which can be changed to increase the size of the system heap. For more information, refer to the section about the INIT Resource 31 in *Inside Macintosh*, Volume 5, "System Startup Information".

Initialization

Figure 10-2 lists the initialization process for the Forwarder, the server, and the client respectively.

MSC NNNN
ART: NN x 17 pi
20.5 pi text to FN b/b

Figure 10-2
Initialization process using the Forwarder



normal processing follows...

Fig.10-2 -COMP (L16)
MCP Developer's Guide
Apple Computer, Inc.
JOYCE ZAVARRO
Illustrator 88
GEORGE M. VRANA

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 a MR-DOS 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 NBP `Look_Up` call.

Normal processing using 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.

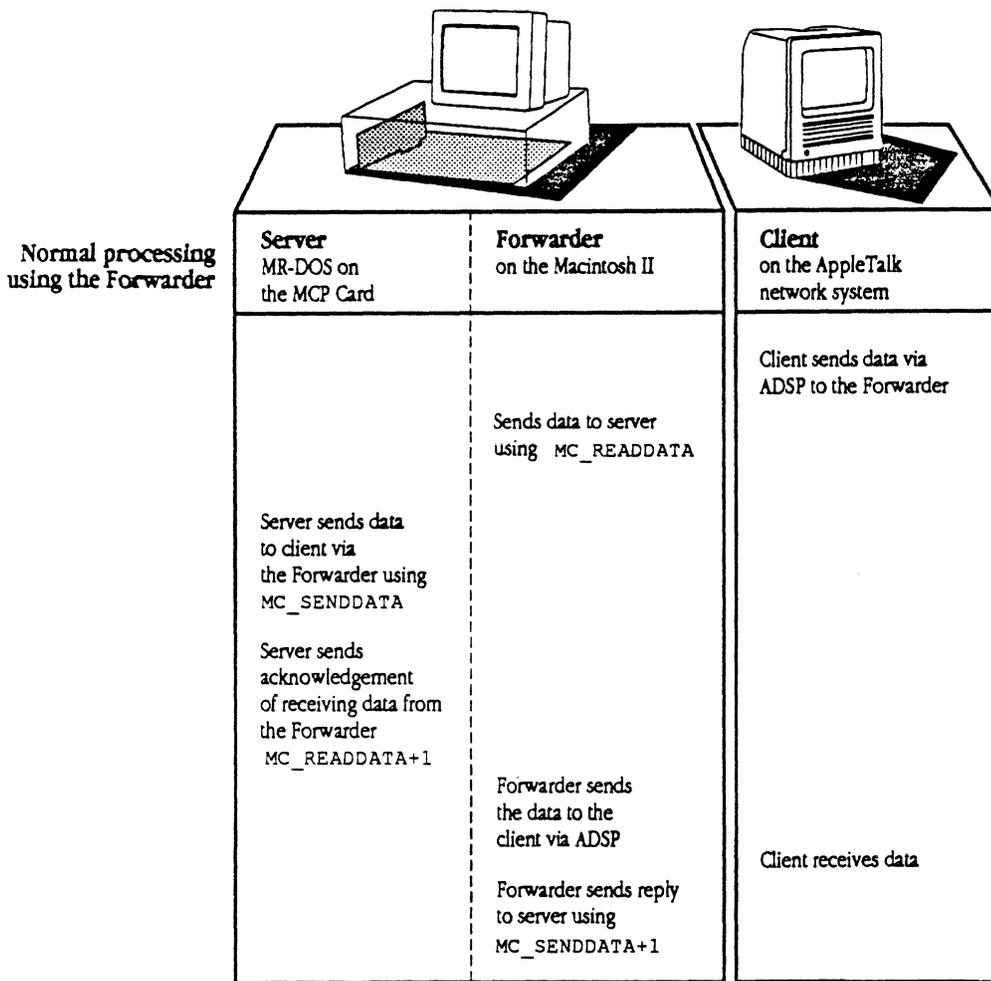
MSC NNNN
ART: NN x 17 pi
20.5 pi text to FN b/b

Figure 10-3
Normal processing using the Forwarder

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.



end of processing follows...

Fig 10-3 -COMP (L16)
MCP Developer's Guide
Apple Computer, Inc.
JOYCE ZAVARRO
Illustrator 88
GEORGE M. VRANA

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.

Completing communication with the Forwarder

Figure 10-4 shows how the client completes communication and terminates the connection.

- ❖ *Note:* In actuality, the server and Forwarder wait continuously for more connections from other clients.

MSC NNNN
ART: NN x 17 pi
20.5 pi text to FN b/b

Figure 10-4 End of processing using the Forwarder

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 availability 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.

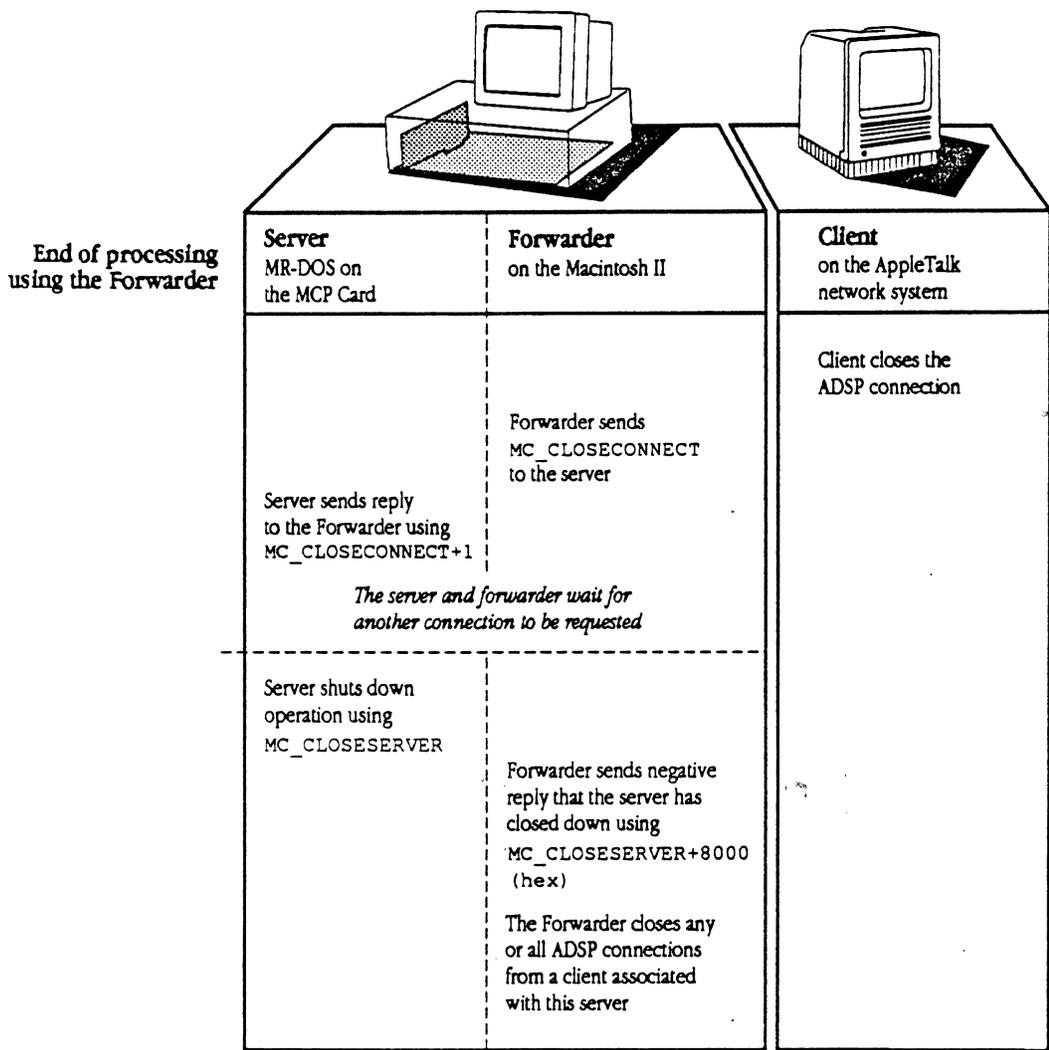


Fig.10-4 -COMP (LI6)
MCP Developer's Guide
Apple Computer, Inc.
JOYCE ZAVARRO
Illustrator 88
GEORGE M. VRANA

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 Apple IPC 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 Apple IPC 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 connection 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:

<code>mCode</code>	<code>MC_OPENSERVER</code>
<code>mFrom</code>	Server's task ID (used by Forwarder to uniquely identify the servers)
<code>mDataPtr</code>	Pointer to NBP EntityName structure with type, object, and zone filled in
<code>mDataSize</code>	size of EntityName structure

The reply parameters for `MC_OPENSERVER` are as follows:

<code>mCode</code>	<code>MC_OPENSERVER + 1</code>
<code>mStatus</code>	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:

<code>mCode</code>	<code>MC_READDATA</code>
<code>mFrom</code>	Forwarder's task ID
<code>mDataPtr</code>	Pointer to data
<code>mDataSize</code>	Length of data
<code>mOData[0]</code>	Connection ID
<code>mOData[1]</code>	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:

<code>mCode</code>	<code>MC_READDATA + 1</code>
<code>mOData[0]</code>	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:

<code>mCode</code>	<code>MC_SENDDATA</code>
<code>mFrom</code>	Server's task ID
<code>mDataPtr</code>	Pointer to data
<code>mDataSize</code>	Length of data (limited to <code>DATA_BUFFER</code> , which is 580 bytes)
<code>mOData[0]</code>	Connection ID
<code>mOData[1]</code>	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:

<code>mCode</code>	<code>MC_SENDDATA + 1</code>
<code>mOData[0]</code>	Connection ID
<code>mStatus</code>	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       -       MR-DOS 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 "mrdos.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 OPENSERVR request to forwarder */

msg = GetMsg();
msg->mTo = fwd_tid;
msg->mCode = MC_OPENSERVR;
msg->mDataPtr = &ent;
msg->mDataSize = sizeof(EntityName);
mid = msg->mId;
Send( msg );
msg = Receive( 0, 0, MC_OPENSERVR+1, 0 );
if (msg->mStatus)
{
    printf("MC_OPENSERVR 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 = GetMessage();
    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.)

```

/*
 *   FWDEExample.c      -      MPW Tool to access "server" on MCP card.
 *
 *   Copyright © 1988, Apple Computer, Inc.  All rights reserved.
 *
 *   The tool is accessed by:
 *
 *   FWDEExample  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          "# FWDEExample 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 */
DSPPParamBlock 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(DSPPParamblock));
    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 when using the Forwarder

The following shows the flow of MR-DOS messages between a typical 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    mData[0] = 00000000    mData[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    mData[0] = 00000000    mData[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   mData[0] = 00000000   mData[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   mData[0] = 00000000   mData[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   mData[0] = 00000000   mData[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   mData[0] = 00000000   mData[1] = 00000000
mSData[2] = 00000000   mOData[0] = 00000000   mOData[1] = 00000000
mOData[2] = 00000000

```

FWD example finished!

Errors returned by the Forwarder

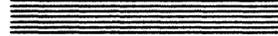
Table 10-2 lists the errors returned by the Forwarder, and briefly describes each.

Table 10-2
Errors returned by the Forwarder

Error	Description
<code>FWE_DupServer</code>	Only one server can be opened per MR-DOS task. (Attempted to open more than one.)
<code>FWE_NoServer</code>	Forwarder did not find a server registered under the current task ID.
<code>FWE_Write</code>	Attempted to issue an <code>MC_SENDDATA</code> before the Forwarder was finished processing the previously issued <code>MC_SENDDATA</code> for this connection.
<code>FWE_NoConnect</code>	The connection ID specified was not found.
<code>FWE_Overflow</code>	The maximum data size of 580 bytes (<code>DATA_BUFFER</code>) was exceeded by <code>MC_SENDDATA</code> .
<code>FWE_NoSMemory</code>	Could not get memory on the Macintosh II to open server.
<code>FWE_NoSListen</code>	The ADSP listener failed.
<code>FWE_NoRegister</code>	The NBP name registration failed. (This error most likely occurred due to a duplicate server name, or improperly-filled in <code>EntityName</code> structure.)



Chapter 11



Troubleshooting Guide

This chapter describes the illegal instructions and debugger calls that can occur when using MR-DOS and the Apple IPC driver, and lists error codes and messages that may be returned for both MR-DOS and the Apple IPC driver.

- ❖ *Note:* This chapter assumes you have a working knowledge of the M68000 microprocessor architecture and instruction set.

What happened?

During development, you will likely encounter crashes or hangs from time to time. Here's what to do when either of those situations occur:

- ❑ To determine the possible cause of a system crash, look at the load map of the code executing on the card, or at the supervisor stack for the Macintosh II.
- ❑ When 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.

Troubleshooting MR-DOS

If the operating system code on the MCP smart card appears to have stopped running, MR-DOS 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 MR-DOS kernel is still functioning. The value `gCommon.gMajorTick` is the major tick counter within MR-DOS, 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 MR-DOS Hangs called "gCommon.gMajorTick Is Incrementing".

If `gCommon.gMajorTick` is not incrementing, MR-DOS may either be 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 MR-DOS has crashed.

On execution of an exception or hardware error interrupt, a MR-DOS 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 MR-DOS Crashes. Otherwise, the system is hung; go to the section on MR-DOS 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 MR-DOS operating system.

`Dumpcard` dumps the card and formats the output to the standard output you specify in MPW. There are two versions of `dumpcard`; the version for the MCP smart card is found in the file `:MR-DOS:Examples:MCP:`. The version for the AST card is found in the folder `:MR-DOS:Examples:AST_ICP:`.

The syntax of the dumpcard tool is

```
dumpcard[-d fwa lwa][-e][-h][-m][-p][-r][-s s1 s2...sn][-t][-v]
```

where	-d fwa lwa	dumps the context of card memory from "fwa" to "lwa" (fwa/lwa in hex)
	-e	do not display 68000 exception vectors
	-h	try to halt processor (if not already halted)
	-m	do not display the task's message queue
	-r	do not display registers (if processor halted)
	-s s1 s2...sn	dump only the cards in the specified slots (the default is all cards found)
	-t	do not dump task tables
	-v	display version information

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 :MR-DOS: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).

```
***** Slot #B
Unable to display registers - processor running
```

```
Initial A5 value .....FB00807C
Memory buffer list ptr..FB01CF48
Slot address.....FB000000
Slot number.....0B000000
Time Base.....A014801C
Major Tick.....00001265
CAP (Magic Number).....1CCA1940
CAP (Pointer).....FB0009CC
CAP (Checksum).....17CA230C
Free message list.....FB0706E6
Unique Counter.....FB000037
Tick Chain.....FB00BB32
Idle Chain.....FB00BB38
Current Task Pointer....FB07EAA8
Idle Loop Counter.....0011E562
Task Table Pointer.....FB07EAF8
Error status.....00000000
```

```

Timeout queue.....FB070508
Priority Table Pointer..FB07EF00
Priority List.....FB07EFF8
Name Unregister pointer.FB0080C6
FWA of message area....FB0705C0
LWA of message area....FB07E850
Initial PC.....FB00C83E
FWA of initial code....FB0086A8
LWA of initial code....FB01CF48
Minor Tick Counter.....00000000
Debugger Pointer.....00000000
Debugger Comm Area.....00000000
Release version.....0100
Current Task ID.....0B000000
Minor / Major Cycle.....0008
Ticks per Second.....0013
Major Cycles deferred....0E02
Major Cycles skipped....0000
Page Latch.....FB00
Name Task TID.....0B000001
ICC Task TID.....0B000003
Trace Task TID.....00000000
Timer Task TID.....0B000002
Messages discarded.....00
Major Flag.....00
Time Queue Flag.....00
Debugger Flag.....00

```

Task Table dump

```

** Task #0 (TCB = FB07EAA8)
Next Task (priority)...FB07EAA8
Next Task (timeout)....00000000
Stack Buffer.....FB07E858
Heap Buffer.....00000000
Program Counter.....FB0086AE
Stack Pointer.....FB07EA60
Code Segment.....00000000
Data Segment.....00000000
Start Parameters.....00000000
Parent TID.....00000000
Status Register.....0004
Page Latch.....FB00
Priority.....00
Status.....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)
    
```

```

FB07EA60: 00 00 00 01 00 00 00 01 00 00 FF FF 00 00 00 00 .....
FB07EA70: FF 00 00 00 00 00 12 77 00 00 FF FF 00 00 00 0D .....w.....
FB07EA80: FB 07 EA A8 FB 00 09 E6 00 AE 8F 20 FB 00 0A 90 .....
FB07EA90: FB 00 09 CC FB 00 80 7C 00 00 00 00 FB 00 8C 2C .....|.....,
FB07EAA0: FB 07 EA F0 01 00 00 0A FB 07 EA A8 00 00 00 00 .....
FB07EAB0: FB 07 E8 58 00 00 00 00 FB 00 86 AE FB 07 EA 60 ...X.....
FB07EAC0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
FB07EAD0: 00 04 FB 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
FB07EAE0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
FB07EAF0: FB 07 EE F8 01 00 00 81 FB 07 EA A8 FB 07 05 08 .....
FB07EB00: FB 06 F4 00 FB 06 E2 F8 FB 06 E1 70 FB 06 D0 68 .....p...h
FB07EB10: FB 06 CE E0 FB 06 B9 D0 00 00 00 00 00 00 00 00 .....
FB07EB20: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
FB07EB30: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
FB07EB40: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
FB07EB50: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 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 (sdos.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.iChecksum	gCommon	mrDOS.h
CAP (Magic Number)	gCAP.iMagic	gCommon	mrDOS.h
CAP (Pointer)	gCAP.iPointer	gCommon	mrDOS.h
Code	mCode	mMessage	os.h
Code Segment	pCodeSeg	pTaskSave	os.h
Current Task ID	gTID	gCommon	mrDOS.h
Current Task Pointer	gCurrTask	gCommon	mrDOS.h
Data Pointer	mDataPtr	mMessage	os.h
Data Segment	pDataSeg	pTaskSave	os.h
Data Size	mDataSize	mMessage	os.h
Debugger Comm Area	gDebugCom	gCommon	mrDOS.h
Debugger Flag	gDebugOn	gCommon	mrDOS.h
Debugger Pointer	gDebugTemp	gCommon	mrDOS.h
Error status	gError	gCommon	mrDOS.h
Free message list	gMsgFree	gCommon	mrDOS.h
FWA of initial code	gFwaCode	gCommon	mrDOS.h
FWA of message area	gFwaMess	gCommon	mrDOS.h
From	mFrom	mMessage	os.h

Heap Buffer	pHeapBuf	pTaskSave	os.h
ICC Task TID	gIccTask	gCommon	mrDOS.h
ID	mId	mMessage	os.h
Idle Chain	gIdleChain	gCommon	mrDOS.h
Idle Loop Counter	gIdleLoop	gCommon	mrDOS.h
Initial A5 value	gInitA5	gCommon	mrDOS.h
Initial PC	gInitPC	gCommon	mrDOS.h
LWA of initial code	gLwaCode	gCommon	mrDOS.h
LWA of message area	gLwaMess	gCommon	mrDOS.h
Major Cycles deferred	gMajorDefer	gCommon	mrDOS.h
Major Cycles skipped	gMajorSkip	gCommon	mrDOS.h
Major Flag	gMajorFlag	gCommon	mrDOS.h
Major Tick	gMajorTick	gCommon	mrDOS.h
Memory buffer list ptr	gBuffList	gCommon	mrDOS.h
Message Q Head	pQHead	pTaskSave	os.h
Message Q Tail	pQTail	pTaskSave	os.h
Messages discarded	gMsgBucket	gCommon	mrDOS.h
Minor / Major Cycle	gMinPerMaj	gCommon	mrDOS.h
Minor Tick Counter	gMinorTick	gCommon	mrDOS.h
Name Task TID	gNameTask	gCommon	mrDOS.h
Name Unregister pointer	gUnregTask	gCommon	mrDOS.h
Next	mNext	mMessage	os.h
Next Task (priority)	pNextTask	pTaskSave	os.h
Next Task (timeout)	pNextTime	pTaskSave	os.h
Other Data	mOData[0]	mMessage	os.h
	mOData[1]	mMessage	os.h
	mOData[2]	mMessage	os.h
Page Latch	gPageLatch	gCommon	mrDOS.h
Page Latch	pPageLatch	pTaskSave	os.h
Parent TID	pParentTID	pTaskSave	os.h
Priority	mPriority	mMessage	os.h

Priority	pPriority	pTaskSave	os.h
Priority List	gPriList	gCommon	mrdos.h
Priority Table Pointer	gPriTable	gCommon	mrdos.h
Program Counter	pPcSave	pTaskSave	os.h
Release version	gRelease	gCommon	mrdos.h
Sender Data	mSData [0]	mMessage	os.h
	mSData [1]	mMessage	os.h
	mSData [2]	mMessage	os.h
Slot address	gSlotAdd	gCommon	mrdos.h
Slot number	gSlotNum	gCommon	mrdos.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	mrdos.h
Tick Chain	gTickChain	gCommon	mrdos.h
Ticks per Second	gTickPerSec	gCommon	mrdos.h
Time Base	gTimeBase	gCommon	mrdos.h
Timeout queue	gTimeout	gCommon	mrdos.h
Time Queue Flag	gTQFlag	gCommon	mrdos.h
Timer Task TID	gTimerTask	gCommon	mrdos.h
To	mTo	mMessage	os.h
Trace Task TID	gTraceTask	gCommon	mrdos.h
Unique Counter	gUnique	gCommon	mrdos.h

MR-DOS crashes

The following steps will help you determine if MR-DOS has crashed and aid in finding the error:

1. First check location \$64A to make sure that `Flag` is non-zero; that means MR-DOS has crashed.
2. If MR-DOS 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 MR-DOS does *not* detect an error, use the loop map (described in the next section) to determine what code was executing at the time of the error. If MR-DOS *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 MR-DOS in the load map are produced by the linker (refer to `makefile` in the file `MCP:Examples:MR-DOS:` for an example). The location `gCommon.gFwaCode` contains the first word address of the code containing the MR-DOS 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 MR-DOS, 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 MR-DOS 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 MR-DOS, 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 MR-DOS error codes

If the MR-DOS 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, MR-DOS 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 MR-DOS

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

\$8000009	eFMSG	tfreemsg (ostrap)	Attempt to free bad message buffer
\$8000008	eMEMB	tfreemem (ostrap)	Attempt to free bad memory buffer
\$8000003	eNPTR	pickTask (ostask)	No processes to run
\$8000007	eOVFL	saveTask (ostask)	Stack overflow detected
\$800000A	eSMMSG	tscnd (ostrap)	Attempt to send bad message buffer
\$800000D	eSTPI	tstoptask (ostrap)	StopTask cannot be called from interrupt
\$800000C	eSTTI	tstarttask (ostrap)	StartTask cannot be called from interrupt
\$800000B	eTIMQ	tscnd (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 MR-DOS 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 MR-DOS. 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 MR-DOS message buffer pool.

The main routine calls the `osinit` routine to initialize MR-DOS. 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 MR-DOS Priority Table.

The main routine calls the `osinit` routine to initialize MR-DOS. 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 MR-DOS process table.

The main routine calls the `osinit` routine to initialize MR-DOS. 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 MR-DOS 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):

$$\text{bHeader.bNext} = \text{bHeader.bSize} * 8 + \text{bHeader}$$

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 MR-DOS 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 MR-DOS 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

eMSG — 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 eMSG 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 MR-DOS 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 MR-DOS data structures and investigate for MR-DOS 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

❖ *Note:* 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 MR-DOS 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.

MR-DOS 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 MR-DOS 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 tool Dumpcard 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. MR-DOS 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.

A task may be 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.
- Examine the bit `psBlock` in `pTaskSave.pStatus`. This bit is set 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.
- Examine the bit `psMany` in `pTaskSave.pStatus`. This bit is clear and the bit `psBlock` is set 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 when the bit `psMany` is clear. 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 `:'MR-DOS IPC':iccmDefs.a` and `:'MR-DOS IPC':iccmDefs.h` 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.Addr`, and `ca_Rec.Ptr` are indexed by the slot number of the card. The Macintosh II is treated as slot 0.

`ca_Rec.Addr` 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.Addr` field in either card be zero.

`ca_Rec.Ptr` is an array of pointers to message buffers. When two cards are communicating, the respective `ca_Rec.Ptr` should contain the addresses of message buffers. If they do not, a card may have run out of message buffers.

MR-DOS may have 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 MR-DOS 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.

A task may be 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.

A task may be 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.

Code on the Idle Chain may be 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 Apple IPC

This section describes the following events that can occur during Apple IPC 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 Apple IPC 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 Apple IPC driver. The first type is more informative and provides error codes or messages (listed in the following tables). When the Apple IPC 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 Apple IPC driver error codes returned from an Apple IPC `Receive` request. Each of these codes is described, along with a potential solution.

Table 11-4
Error codes for Apple IPC driver

Value	Name	Explanation
-64	NoQueueErr	No more queues available
-65	QueueBusy	<code>Receive</code> already outstanding on queue

Table 11-5 lists the possible error messages from the INIT resource that installs the Apple IPC driver. Each of these messages is described in this section, along with a potential solution.

Apple IPC crashes

This section describes the crashes that can occur with Apple IPC because of improper parameter usage; corruption of either the Apple IPC driver or its internal data structures; corruption of the Apple IPC internal data structures during request execution or periodic processing; or during invocation of the Apple IPC driver or the Apple IPC Name Manager.

Crashes during Macintosh II startup

During Macintosh II startup, an INIT31 resource found in the Apple IPC file installs the Apple IPC driver and the Apple IPC 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.

Apple IPC INIT31 — Unit Table full

Description: 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 Apple IPC file or remove another driver.

Apple IPC INIT31 — No DRVR resource in file

Description: INIT31 executes a `DebugStr` call if it does not find a driver of resource type 'DRVR' and resource name '.IPC' in the Apple IPC file. This indicates that the Apple IPC file is in error (due to improper file generation or data corruption).

Solution: Boot from another system disk and replace the 'Apple IPC' file.

Apple IPC INIT31 — Failed to open driver

Description: INIT31 executes a `DebugStr` call if the Apple IPC driver cannot be opened successfully. This indicates there is a serious problem either with the Apple IPC driver or with the Macintosh II operating system.

Solution: Boot from another system disk and replace the 'Apple IPC' file.

Crashes with improper parameter usage

This section describes the events relating to improper parameter usage that can cause the Apple IPC driver to crash. These crashes occur when the Apple IPC driver detects a bad message pointer passed as a parameter to a driver request.

The Apple IPC 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 Apple IPC queue
- `0xFF` if ICCM has obtained a message for internal use

Apple IPC FreeMsg — Bad message pointer

Description: The Apple IPC driver executes a `DebugStr` call if user code invokes a `FreeMsg` request with a bad message pointer.

Solution: Diagnose problem, correct code, and retry.

Apple IPC Send — Bad message pointer or mFrom

Description: The Apple IPC driver executes a `DebugStr` call if user code invokes a `Send` request with a bad message pointer.

Solution: Diagnose problem, correct code, and retry.

Crashes during driver initialization

The Apple IPC driver and the Apple IPC Name Manager can cause a crash due to detection of data corruption during their initialization sequences.

Apple IPC — Missing resource: Apple IPC entries

Description: The Apple IPC driver executes a `DebugStr` call if it does not find a resource of type `'aipn'` and name `'Apple IPC Entries'` in the Apple IPC file, indicating that the Apple IPC file was either improperly generated or corrupted after generation.

Solution: Boot from another system disk and either replace the Apple IPC file or add the missing resource to the Apple IPC 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`).

Apple IPC — Unable to get space from system heap

Description: At startup, the Apple IPC driver executes a `DebugStr` call if it cannot allocate a 12-byte non-relocatable 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.

Apple IPC Name Manager — Missing `aipn` resource: `NameManagerEntries`

Description: The Apple IPC Name Manager executes a `DebugStr` call if it does not find a resource of type `'aipn'` and name `'NameManagerEntries'` in the Apple IPC file, indicating that the Apple IPC 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 Apple IPC file using `ResEdit`. The resource consists of a 16-bit word indicating the number of entries allowed in the Name Manager's tables (the initial value is `0x0012`).

IPC driver crashes during execution

The following events cause a crash if the Apple IPC 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`

Apple IPC `KillReceive/CloseQueue` — timeout queue error

Description: The Apple IPC driver executes a `DebugStr` call if there is an 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.

Apple IPC `Send` — timeout queue error

Description: 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 Apple IPC driver executes a `DebugStr` call.

Solution: Report the problem to Apple Developer Services.

Apple IPC Periodic processing — timeout queue error

Description: The Apple IPC 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 Apple IPC 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.

Apple IPC Receive — timeout queue error

Description: The Apple IPC 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.

Apple IPC Receive — Interrupt routine did blocking Receive

Description: The Apple IPC 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

Description: If the Apple IPC 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

Description: If the Apple IPC 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

Description: If the Apple IPC 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.

IPC glue code crashes

This section describes troubleshooting guidelines for Macintosh II applications using the Apple IPC driver.

Requests to the Apple IPC 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 Apple IPC 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.

Apple IPC hangs

The Macintosh II may appear to be hung while executing Apple IPC request code.

Events causing Apple IPC hangs

The following sections describe two of the most common events that could result in this condition.

Macintosh II 32-bit mode debugger hang

Description: The Apple IPC 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.

Unsatisfied blocking Receive request

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 Apple IPC 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 Apple IPC driver returns a zero message pointer should the time specified elapse.

Examining the Apple IPC global area

You can examine the Apple IPC global area to determine the state of a task on the Macintosh II that is using the Apple IPC 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 'Apple IPC' folder.

Examine the Apple IPC 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

Finding the Apple IPC global area

There are many ways to find the Apple IPC global area; these methods get progressively more complicated, but yield the same results. Therefore, only two of these methods are discussed here.

The first method is the most simple and easiest to use. Simply issue a `GetIPCg` request to the Apple IPC driver. The driver returns the starting address of the Apple IPC global area.



Part III



Hardware Development

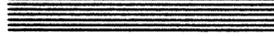
Part III, Hardware Development, provides:

- an introduction to the MCP card
- descriptions of the hardware specifications
- overview of NuBus on the MCP card, with functional examples





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.

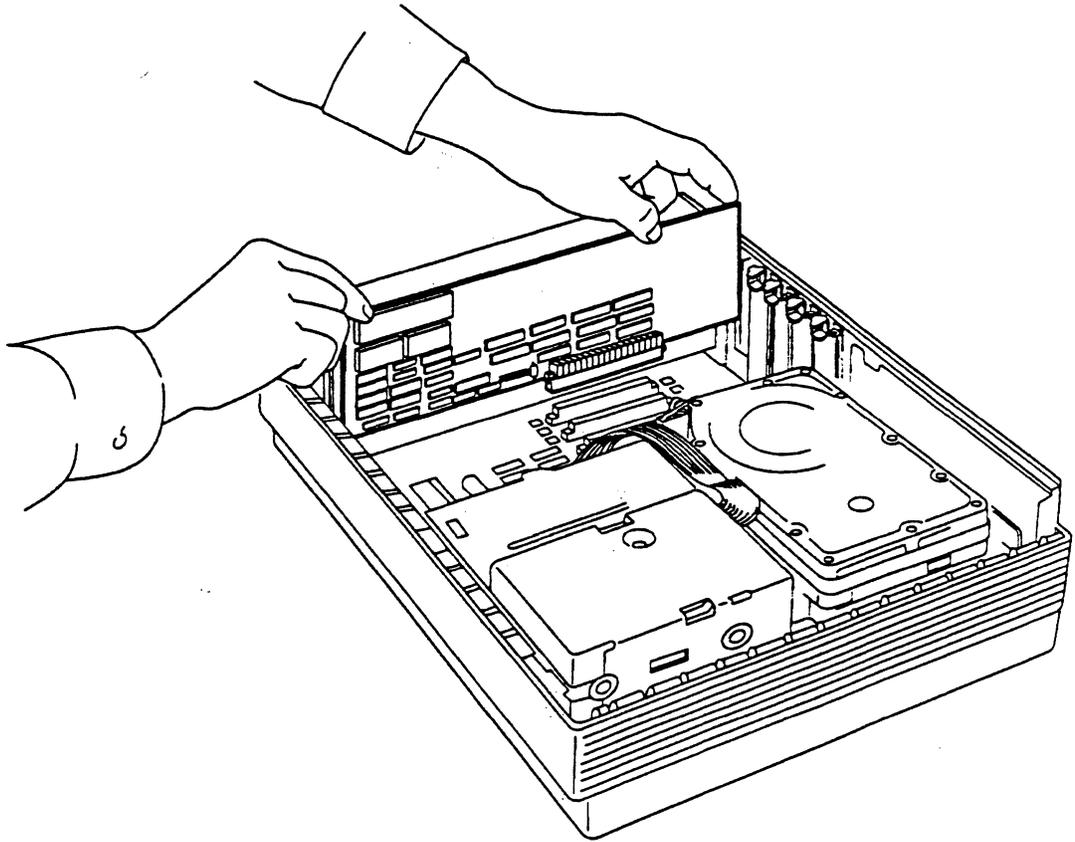
Introduction to the MCP card

Apple Computer has developed 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.

MSC NNNN
ART: NN x 8.5 pi
11 pi text to FN b/b

Figure 12-1
MCP card



*Fig. 12-1 -COMP (L11)
MCP Developer's Guide
Apple Computer, Inc.
JOYCE ZAVARRO
Illustrator 88
GEORGE M. VRANA*

Hardware 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

Processor

The I/O Processor utilizes a 10 Megahertz 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 ROMS, yielding a 64-Kilobyte 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.

- ❖ *Note:* Refer to the appropriate sections in Chapter 16 for information on required resources for the MCP card and adding code to the ROM.

RAM

The card contains 1/2 megabyte 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-EFFFFFF	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-BFFFFFF	NuBus
800000-9FFFFFF	I/O Interface Logic
400000-7FFFFFF	I/O Interface Logic
080000-3FFFFFF	Future RAM
000000-07FFFF	RAM (with 1/2 MB of RAM)

Timer

The MCP card provides an internal 6.5536 millisecond timer. Every 6.5536 ms, a level 1 interrupt occurs. This interrupt is cleared by reading location C00002.

❖ *Note:* 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 FXXXXXX will place the 68000 in RESET, and any access to CXXXXXX 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-FFFFFF).

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 12-3 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 12-4 shows an example of a simple NuBus slave design, with explanatory notes.

Figure 12-5 shows the read and write timing cycles for the simple NuBus slave design shown in Figure 12-4.

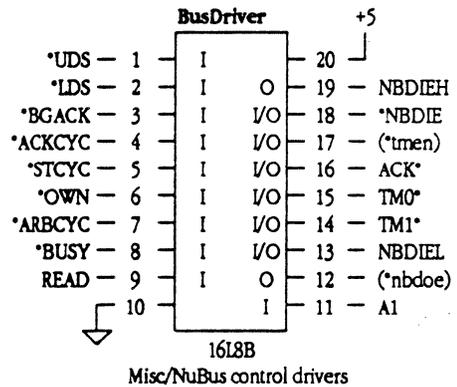
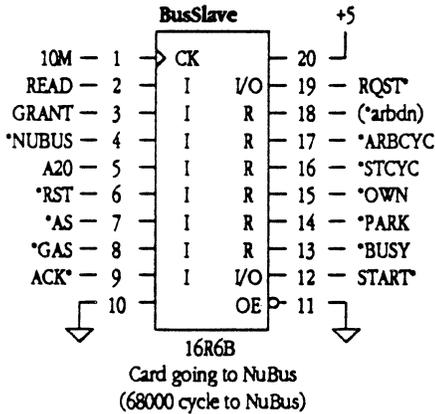
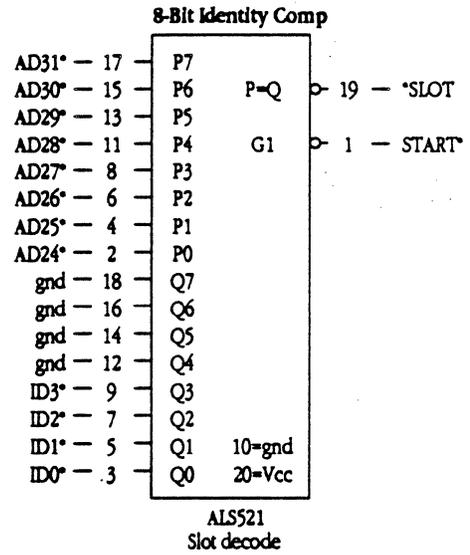
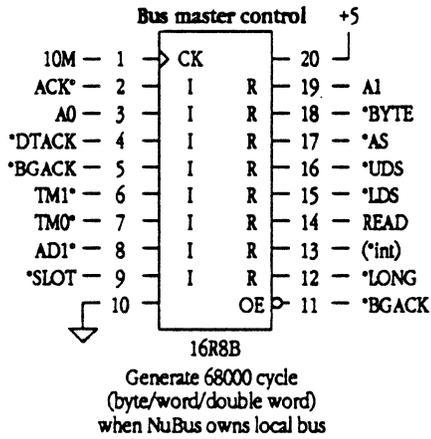
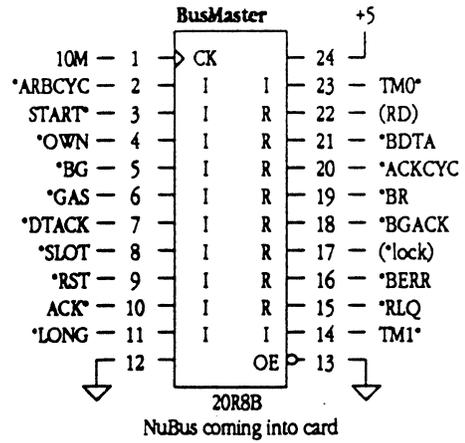
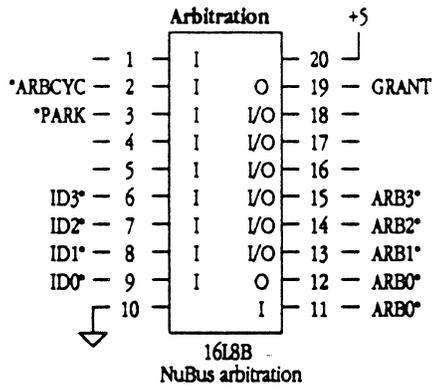


Fig. 12-2 -COMP (L13)
MCP Developer's Guide
Apple Computer, Inc.
JOYCE ZAVARRO
Illustrator 88
GEORGE M. VRANA

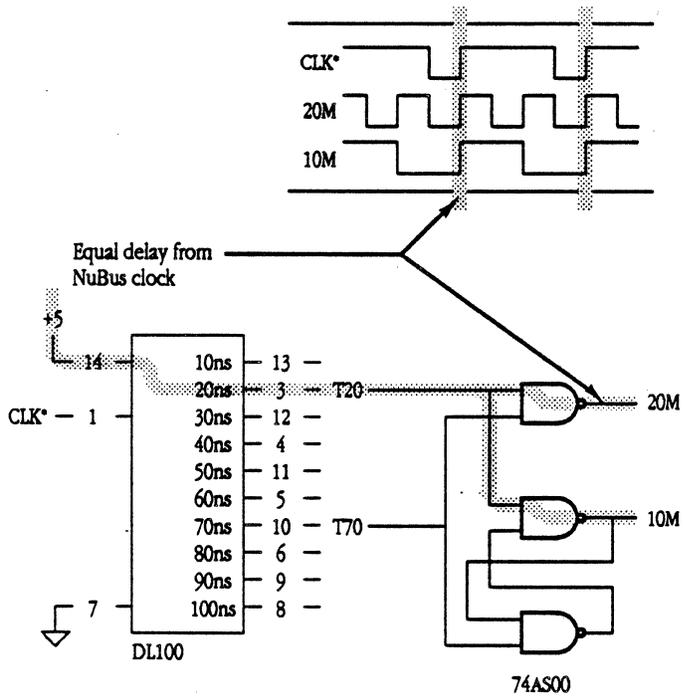
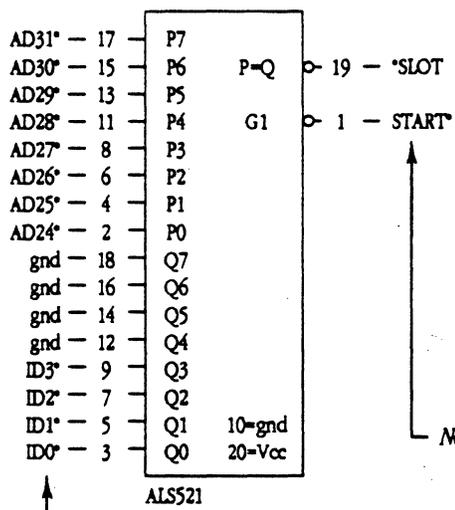


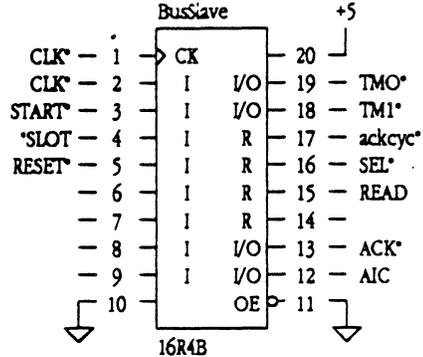
Fig. 12-3 -COMP (L)
MCP Developer's Guide
Apple Computer, Inc.
JOYCE ZAVARRO
Illustrator 88
GEORGE M. VRANA



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)

Note: Run START in here if you are not using it anywhere else



```

* 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

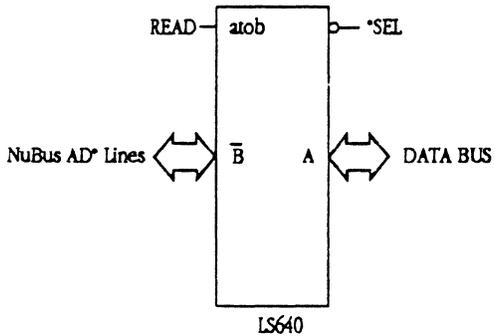
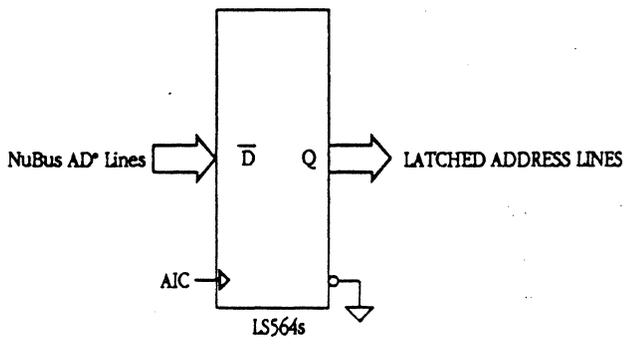
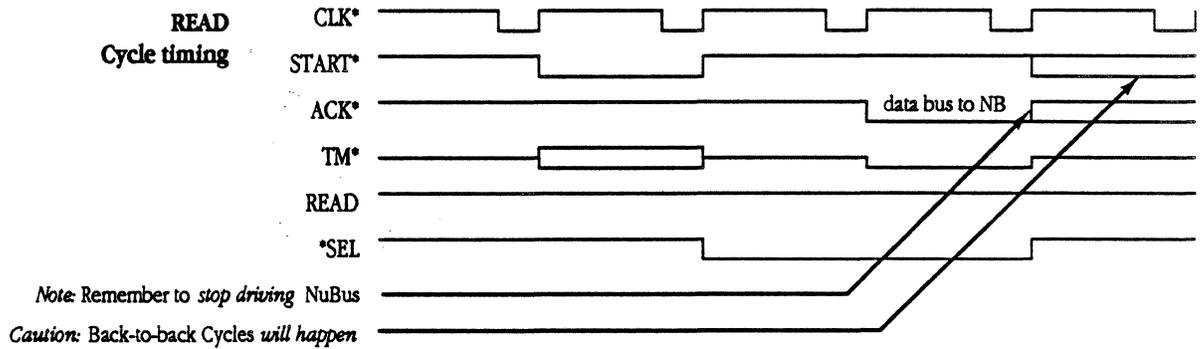
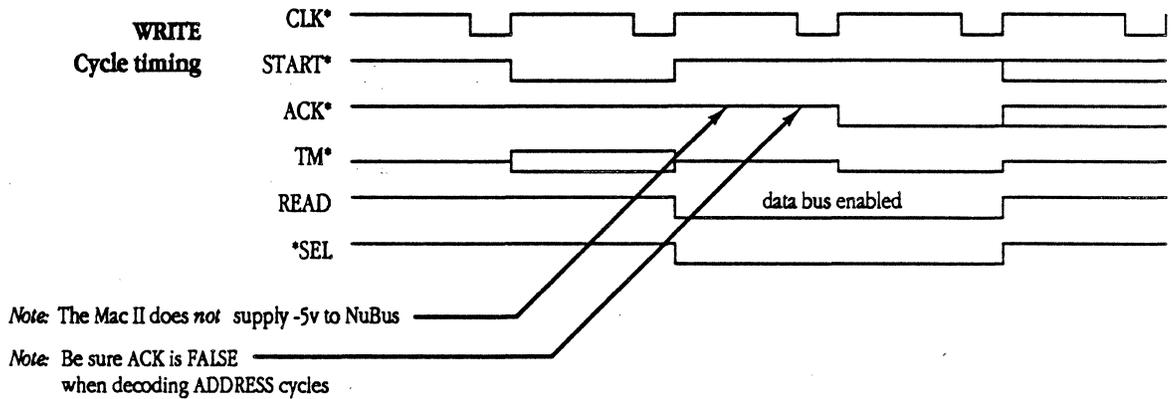


Fig. 12-4 (L-14)
MCP Developer's Guide
Apple Computer, Inc.
JOYCE ZAVARRO
Illustrator 88
GEORGE M. VRANA



READ: TM1=0
 ACK: TM0=1
 TM1=1

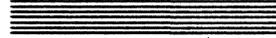


WRITE: TM1=1
 ACK: TM0=1
 TM1=1

Fig. 12-5 -COMP (L-15)
 MCP Developer's Guide
 Apple Computer, Inc.
 JOYCE ZAVARRO
 Illustrator 88
 GEORGE M. VRANA



Chapter 13



Lists for the MCP Card

This chapter provides listings for the PAL equations and a parts list for the MCP card.

- ❖ *Note:* The latest schematics for the MCP card are enclosed as separate pages at the back of this document.

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  /NBDIE 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}
          ;
    NBDIE = 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          BMct1          (6303)

        DATE:             9/30/87
        VERSION:          1A

.NAMES
    10M /ACK  A0  /DTACK /BGACK  /TM1  /TMO /AD1  /SLOT  GND
    en  /LONG /int  READ  /LDS   /UDS  /AS  /byte A1   VCC

.EQUATIONS

    byte := SLOT * TMO * /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 rea
      + 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 1
      + 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 AC
      ;
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 faile
      ;
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 faile
      ;
```

```

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
+ /arbdn * STCYC * BUSY * /GAS           {else c
;
.IF ( OWN )          START = STCYC * BUSY           {Drive START while we own bus}
+ STCYC * /arbdn    {assert during STCYC except IDLE c
;
.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            {000000-3FFFFFF - ROM w/ setup}
      + AS * A23 * A22 * /A21 * /A20        {C00000-CFFFFFF - ctl reg}
      + GAS * DDTA * A23                    {good hold, not RAM}
      + AS * /A23 * /A22 * /setup * /RFCYC  {000000-3FFFFFF - RAM}
      + GAS * DDTA * /RFCYC                 {good hold, RAM, for rmw}
      ;
.IF ( GAS * FC0 * FC1 (* /NBACK) )
VPA  = AS
      ;
NUBUS = A23 * /A22 * A21                      {NuBus = A00000-BFFFFFF}
      ;

.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 /tmdly /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
            ;
            tmdly := TMR
            ;
            RAO = MUX * A2
                  + /MUX * A1
            ;
            TMRIR := TMR * /tmdly * /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          (always 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                          {always 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 13-2
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%

(continued)

Table 13-2 (cont'd)

Quantity	Name	Description
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)



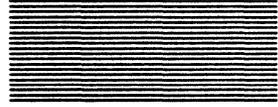
Part IV



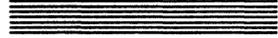
MCP Diagnostics

Part IV, MCP Diagnostics, describes:

- the three levels of diagnostics provided for use with the MCP card
- the menus for the MCP_Diagnostics application and a description of each test available in the application
- the commands and routines used to create a customized diagnostic application



Chapter 14



Diagnostics for the MCP Card

This chapter provides an overview of the diagnostic software provided for the Macintosh Coprocessor Platform card, and describes the resources provided on the MCP distribution disk. This chapter also describes the initial power-up diagnostic tests.

Warning

Some of the user interface and menus will change in future versions. However, any hardware tests you write that use the MCP_Diagnostic application should remain compatible. Please refer to the file :Sequencer:Errata on the MCP_Diagnostics disk for more information.

What does Apple provide?

Apple Computer provides both firmware and software. The software is provided on the MCP diagnostics distribution disk, and the firmware is provided both on the MCP card declaration ROM and the distribution disk.

The diagnostic tools are supplied with the card; you can create additional diagnostic tools using the information provided in this guide and on the distribution disk.

- ❖ *Note:* Apple Computer provides these diagnostics solely as a framework for test verification of board designs, and does not guarantee these tests to be exhaustive.

As this diagnostic is software based, some aspects of hardware verification cannot be assured in an office environment. For example, adding test equipment such as a logic analyzer and an in-circuit emulator may cause marginal cards to fail by loading data, address, and state lines. Heat conditions close to the upper heat limit have also produced failures in otherwise working boards.

The MCP distribution disk provides a libraries of functions as well as a working sample that you can change. Table 14-1 describes the folders provided on the MCP Diagnostics distribution disk.

Table 14-1
Diagnostics folders

Name of folder	Description of contents
Sequencer	Source code and makefile necessary to create sequencer; include files necessary for building all levels of test diagnostics
MCP	The application MCP_Diagnostic, used for both second- and third-level tests, as well as the source code, libraries, and instructions to create your application-specific diagnostic
Level 3 Examples	Sample source code and makefile to create standalone applications that can be downloaded and executed by the 68000 on the MCP card
DiagTool	Source code to an MPW diagnostic tool

❖ *Note:* Refer to Appendix A for a complete description of all files and folders on the MCP distribution disks.

Diagnostic capabilities

There are three levels of diagnostics provided for the MCP card. The first two levels run tests between the Macintosh II and the MCP-based card; the third level is for on-card testing that is reported on the Macintosh II screen. You can use the first two levels of tests as provided, running generic tests for the MCP card, or you can customize the second- and third-level diagnostic tests as applications to be run on the Macintosh II computer.

There is also a sample MPW diagnostic tool named `MCPdiagtool` on the distribution disk to show how to use the diagnostic tests provided in the library `diaglib.o`.

Table 14-2 describes the three levels of diagnostics provided for the MCP card.

Table 14-2
Levels of MCP diagnostics provided

Level	Description	Functions provided
1	ROM diagnostic	<p>Tests basic card functionality, runs from the on-board ROM automatically at power-up and NuBus reset</p> <p>Returns Good/No Good status to the Slot Manager</p>
<p>❖ <i>Note:</i> If a specific test fails, the remaining tests are not executed.</p>		
2	Sequential diagnostic application	<p>Provides more extensive testing than in the power-up tests</p> <p>Returns locations and descriptions of failures</p> <p>Provides hooks and a Macintosh user interface for developer-specific testing</p>
3	Concurrent diagnostic	<p>Downloads and executes non-toolbox calling test applications to MCP-based smart cards</p> <p>Provides an environment for concurrent card testing and data logging</p> <p>Provides MultiFinder background task compatibility</p>

❖ *Note:* Power-on and primary initialization tests indicate what failed by the error code shown in Table 14-4. Because of some dependencies in the diagnostic application, if power-on or primary initialization tests fail, spurious error messages may occur from the diagnostics.

The first diagnostic tool is described in greater detail in this chapter. See Chapter 15 for the second-level (sequential) tests and information on adding code to the ROM for your card-specific application, and Chapter 16 for third-level (concurrent) tests.

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 `App1PowerOn.a` (described in the next section).

Next, the level 1-7 auto vector interrupts are enabled to the routines defined in the file `App1PowerOn.a`. Then the code executes a test reserved for the application you develop. Currently, this is a stub function named `VendorPowerUp` in the file `App1PowerOn.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 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.

❖ *Note:* 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

- a `Write` 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 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.

❖ *Note:* If any application uses this area of memory (such as MR-DOS), 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 14-3 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 14-3
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 14-4 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 14-4
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
-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 power-up.

Using the MCP_Diagnostic library

An example MPW tool named `MCPDiagTool.c` has been created to introduce the use of the MCP diagnostic application and support routines. This tool exercises most of the routines provided in the MCP diagnostic library.

To use this MPW tool:

1. Open the folder named `DiagTool` on the MCP Diagnostics distribution disk.
2. Open the MPW file `MCPDiagTool.c`.
3. Enter the following command string:

```
MCPDiagTool [-(9-E)] [-i NumIterations] [-v]
```

where

`9-E` allows the operator to select specific slots. If you do not specify any slot options, all Macintosh II slots are searched for MCP cards and then tested. For example, to test MCP-based cards in slots A and E, you would enter:

```
MCPDiagTool -a -e
```

`-i` sets the number of iterations through the test sequence. Valid iteration counts range from 1 to 65000. The default number of iterations is once through the entire test sequence.

`-v` prints the name and slot of each test run, and the pass or fail status. The default is to print information on test failures only.

The following listing of the code (from the file `MCPDiagTool.c`) illustrates how to use many of the routines contained in the MCP diagnostic library. This commented source code also shows how these Pascal and assembler diagnostic library routines can be called from the C language.

```

/*****
/*      Copyright © 1988 Apple Computer, Inc.  All rights reserved      */
/*                                                                              */
/* An MPW tool that executes many of the discrete PAL MCP diagnostic routines */
/* in the DiagLib.o library (currently compiled with MPW C 2.0.)          */
/*                                                                              */
/*      Routines requiring slot # expect 0 for slot 9; 5 for slot E      */
/*                                                                              */
/*      Warning: This diagnostic is still under development.              */
/*                                                                              */
*****/

#include <stdio.h>
#include <signal.h>
#include <Types.h>

pascal long TickCount()
    extern Oxa975;
pascal void RotateCursor(tick)          /* Rotates cursor using given counter */
    long tick;
    extern;
pascal void SpinCursor(tick)           /* Rotates cursor using internal counter*/
    short tick;
    extern;

# define      CURSORFORWARD      (32) /* Tell RotateCursor to go forward. */
# define      CURSORBACKWARD    (-32) /* Tell RotateCursor to go backward. */

pascal short GetCPU(); /* returns 0 fo 68000, 1=68010, 2=68020, 3=68030 */
    extern;
pascal void InstallMyErrV() /* install bus error handling vector which calls */
    extern; /* BusErrDialog, a routine found below */
pascal void InstallOldV() /* reinstalls original bus error vector */
    extern;
pascal void SetBusErrRetry(count) /* number of bus errors before */
    /* terminating diagnostic */
    short count;
    extern;
pascal short GetBusErrRetry() /* if result != count from above, # of */
    extern; /* Bus errors = GetBusErrRetry() - count */
pascal short CheckCRC(slot) /* returns 0 if calculated checksum matches */
    short slot; /* longword found in declaration ROM */
    extern;
pascal short GetApplID(slot) /* returns word at card ROM address 0xFFFFE8 */
    short slot; /* a way to determine card app without */
    /* slot manager */
    extern; /* useful in mac II w/ Rev A ROMs */
pascal long GetRAMSize(slot) /* returns size of card ram in bytes */
    short slot;
    extern;

```

```

pascal short TestAndSet(slot, which) /* cycles card and mac processors */
/* through repeated TAS cycles */
short slot; /* reports error if card & system obtain access on any cycle */
short which; /* 0 performs TAS on card memory, 1 = system board */
extern;

pascal short TimeVerify(slot) /* verifies speed of card's timer interrupt */
short slot; /* +/- 10 % */
extern;

pascal short InterComm(mSlot, sSlot) /* does ROM read, RAM modify/read from */
/* mSlot to sSlot */
short mSlot, sSlot;
extern;

pascal int GetVPUS(slot) /* Vendor Power Up status defined on */
/* powerup vendor fail */
short slot; /* returns value from card RAM $134 */
extern;

pascal int GetVPIS(slot) /* Vendor Primary Initialization status */
short slot;
extern; /* returns value from card RAM $138 */

pascal void powerup(slot) /* initiates card ROM powerup code */
short slot; /* status returned 1 sec later by GetPowerUpStatus */
extern;

pascal short GetPowerUpStatus(slot) /* returns card power up code */
short slot; /* see code below for status interpretation */
extern;

pascal short Lv11Init(slot) /* executes & returns card primary init status */
short slot; /* see code below for status interpretation */
extern;

pascal short IsCommCard(slot) /* returns -1 if $C3D2 @ ROM location $FFFFEA */
short slot; /* returns 1 if $C3D2 is not @ ROM location $FFFFEA */
extern; /* returns 0 if bus error occurred accessing $FFFFEA */
/* Note: InstallMyErrV must be called before using routine */

pascal void RAMTest(testNum, beginAddr, lastAddr, pass, failAddr, expected, actual)
short testNum; /* controls which test is run (see code below)*/
long beginAddr, lastAddr; /* 32 bit card address of start and end */
/* of RAM under test */
short *pass; /* if non-zero, the following fields */
/* are returned */
long *failAddr; /* address of failure (not all RAM tests */
/* return this value) */
long *expected; /* expected value */
long *actual; /* actual value */
extern;

pascal short PrimeWrite(slot) /* write & read/verify a word to card RAM */
short slot; /* (not as comprehensive as data line test) */
extern;

pascal short WriteMacLong(slot) /* write & read/verify a long from card */
short slot; /* to Mac RAM (not comprehensive) */
extern;

pascal short Timer(slot) /* test the card's level 1 interrupt */
short slot; /* assumes level 2 interrupt works */
extern;

```

```

pascal short testIOP(slot)      /* Macintosh-to-card interrupt test */
    short slot;
    extern;
pascal short NMRQ(slot)        /* Card-to-Macintosh interrupt test */
    short slot;                /* assumes level 2 interrupt works */
    extern;

pascal void Debug()
    extern 0xa9ff;             /* break into macsbugs */

#define NumTests 12           /* byte inversion not run as it takes soo long */

short      test[6];           /* array of slots to be tested          */
short      verbose = 0;       /* print pass/fail info for every test */
short      iterations = 1;    /* default # of times each test executed */
char       *progName;        /* Name of this program.                */
short      icount;           /* iterations completed count          */
short      busErrV;         /* bus error vector installed flag      */

main(argc, argv)
int      argc;
char    **argv;
{
    unsigned long    wait;
    short            i;
    int              j, slot;
    int              status;
    char             testStr[255];
    long             beginAddr, lastAddr, failAddr, expected, actual;
                                /* RAM test vars */
    SignalMap        sigMap;

    ParseArgs(argc, argv);
    if (GetCPU() < 2)
        ParamDie("This tool incompatible with 68000 or 68010
processors.", " ");

    setbuf(stdout, (char *) 0);          /* don't buffer output */

    InstallMyErrV();                    /* set up bus error handler/find card routine */
    if (!test[0] && !test[1] && !test[2] && !test[3] && !test[4] && !test[5]) {
        for (i=0; i<6; ++i) {
            /* find all cards with $C3D2 @ ROM location $FsFFFEEA */
            if (IsCommCard(i) == -1) {
                test[i] = 1;
                fprintf(stderr, "Testing slot %X\n", i+9);
            }
        }
    }

    InstallOldV();                      /* remove special bus err vector */

    SetBusErrRetry(2);                  /* exit diagnostic after 2nd bus error (1 retry) */
}

```

```

for (i=0; i<iterations; ++i) {
    fprintf(stderr, "Starting pass %d of %d...\n", i+1, iterations);
    for (j=0; j < NumTests; ++j) {
        for (slot=0; slot<6; ++slot) {
            if (test[slot]) {
                RotateCursor(CURSORFORWARD);
                sigMap = sighold(SIGALLSIGS);
                /* disable cmd-. while in 32 bit mode */
                InstallMyErrV();
                /* set up bus error handler/find card routine */
                switch (j) {
                    case 0: /* CRC test on card ROM */
                        strcpy(testStr, "CRC check");
                        status = CheckCRC(slot);
                        break;
                    case 1: /* run powerup code from ROM on the card */
                        strcpy(testStr, "Power-on");
                        powerup(slot);
                        for (wait=(unsigned long)TickCount()+50;
                            wait > (unsigned long)TickCount();)
                            ; /* wait for test to complete */
                        status = GetPowerUpStatus(slot);
                        if (status == -32) status = 0;
                        /* all tests passed */
                        switch (status) {
                            case 0: strcat(testStr, " ROM tests"); break;
                            case 5: strcat(testStr, " test didn't
                                complete:"); break;
                            case -1: strcat(testStr, " data lines test");
                                break;
                            case -14: strcat(testStr, " RAM test"); break;
                            case -2: strcat(testStr, " test pattern read");
                                break;
                            case -6: strcat(testStr, " CRC test"); break;
                            case -16: strcat(testStr, " card application
                                specific test");
                                status = GetVPUS(slot);
                                /* Vendor Power Up Status */
                                if (!status) strcat(" returned
                                    failure status 0, has NOT");
                                break;
                            default: strcat(testStr, " undefined error");
                                break;
                        }
                    case 2: /* run primary init test */
                        strcpy(testStr, "Primary initialization");

                        status = Lvl1Init(slot);
                        switch (status) {
                            case 0: strcat(testStr, " ROM tests");
                                break;
                            case 1: strcat(testStr, " Nubus Write/Read");
                                break;

```

```

        case 2: strcat(testStr, " card reset test");
break;
        case 4: strcat(testStr, " card level 1 interrupt
test");
break;
        case 8: strcat(testStr, " card level 2 interrupt
test");
break;
        case 16: strcat(testStr, " card application
specific test");
status = GetVPIS(slot);
/* Vendor Primary Init Status */
if (!status) strcat(" returned failure status 0,
has NOT");
break;
        default: strcat(testStr, " undefined error");
}
break;
case 3:
strcpy(testStr, "Card to Macintosh (NMRQ) interrupt
test");
status = NMRQ(slot);
break;
case 4:
strcpy(testStr, "Test&Set on Macintosh RAM contest");
status = TestAndSet(slot, 1);
break;
case 5:
strcpy(testStr, "Test&Set on card RAM contest");
status = TestAndSet(slot, 0);
break;
case 6: strcpy(testStr, "RAM data lines test");
goto common;
case 7: strcpy(testStr, "RAM stuck cell test");
goto common;
case 8: strcpy(testStr, "RAM address lines test");
goto common;
case 9: strcpy(testStr, "RAM fixed pattern test");
goto common;
case 10: strcpy(testStr, "RAM marching ones test 1");
goto common;
case 11: strcpy(testStr, "RAM marching ones test 2");
goto common;
case 12: strcpy(testStr, "RAM byte inversion test");
common:
beginAddr = 0xF9000000 + slot*0x1000000;
/* 32 bit addressing */
lastAddr = beginAddr + GetRAMSize(slot);
RAMTest((short)j-5, beginAddr, lastAddr, &status,
&failAddr, &expected, &actual);

if (status)
fprintf(stderr, "Expected %lX, found %lX at
address %lX;", expected, actual, failAddr);

```

```

        break;
    default:
        fprintf(stderr, "Test #d called, but is not
            defined\n", j);
        break;
} /* switch (j) */

InstallOldV(); /* remove special bus err vector */
sigrelease(SIGALLSIGS, sigMap); /* allow cmd-. abort */
if (status) {
    fprintf(stderr, "Slot %X: %s FAILED, ID=%d\n", slot+9,
        testStr, status);
    if (j==1 || j==2)
        fprintf(stderr, "Warning: further error output may
            be caused by the above problem(s).\n");
}
else if (verbose)
    printf("Slot %X: %s passed.\n", slot+9, testStr);
} /* if (test[slot]) */
} /* for (slot=0; slot<6; ++slot) */
} /* for (j=0; j < NumTests; ++j) */
} /* for (i=0; i<iterations; ++i) */
printf("%d pass%scompleted.\n", i, (i==1)?" ": "es ");
}

ParseArgs(argc, argv)
int     argc; /* Count of command arguments. */
char *argv[]; /* Command argument strings. */
{

    progName = argv[0]; /* parse program name */
    ++argv;
    --argc;

    for ( ; argc > 0; --argc, ++argv ) {
        if ( argv[0][0] == '-' ) { /* flags */
            switch ( argv[0][1] ) {
                case '9': test[0] = 1; break;
                case 'a': case 'A': test[1] = 1; break;
                case 'b': case 'B': test[2] = 1; break;
                case 'c': case 'C': test[3] = 1; break;
                case 'd': case 'D': test[4] = 1; break;
                case 'e': case 'E': test[5] = 1; break;
                case 'p': /* Print verbose info. */
                    verbose = 1;
                    break;
                case 'i':
                    if ( --argc <= 0 ) {
                        ParamDie("Missing iteration count after ",
                            argv[0]);
                    }
                    ++argv;
                    iterations = (short) atoi(argv[0]);
            }
        }
    }
}

```

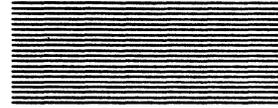
```

        break;
    default:
        fprintf(stderr, "### Usage: %s [- SlotNum(9-E)]
        [ -i NumIterations] [-progress]\n", progName);
        fprintf(stderr, "### %s aborted.\n", progName);
        exit(1);
    } /* switch ( argv[0][1] ) */
} /* if ( argv[0][0] == '-' : */
else {
    fprintf(stderr, "### Usage: %s [- SlotNum(9-E)]
    [ -i NumIterations] [-progress]\n", progName);
    fprintf(stderr, "### %s aborted.\n", progName);
    exit(1);
}
} /* for ( ; argc > 0; --argc, ++argv ) */
}

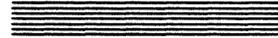
ParamDie(desc1, desc2)
char *desc1; /* 1st half of problem description. */
char *desc2; /* 2nd half of problem description. */
{
    fprintf(stderr, "### %s - %s%s\n", progName, desc1, desc2);
    fprintf(stderr, "### %s aborted.\n", progName);
    exit(1);
}

pascal void BusErrDialog(codeLoc, mode, accessLoc)
long codeLoc;
short mode; /* special status register from exception stack frame */
long accessLoc;
{
    fprintf(stderr, "\n### A bus error occurred executing code at or before
    %lX\n", codeLoc);
    if (mode & 0x40)
        fprintf(stderr, "### Program attempted a read to location
        %lX\n", accessLoc);
    else fprintf(stderr, "### Program attempted a write to location %lX\n",
        accessLoc);
    fprintf(stderr, "### Warning: Further MPW operations may fail. Quit MPW &
        restart to reset environment.\n");
    fprintf(stderr, "### %s aborted.\n", progName);
    InstallOldV(); /* remove bus error handler */
    exit(3);
}

```



Chapter 15



MCP Sequential Diagnostics

This chapter describes the second level of diagnostics for the MCP card and describes how to include application-specific tests as part of the diagnostic. This chapter assumes that you have installed the MCP card and copied to your hard disk the MCP diagnostic software provided on the distribution disk.

An overview

The application called MCP_Diagnostic provides a standard diagnostic user interface and a set of routines that interface the Macintosh II to the core diagnostics of the MCP card's hardware application. This application includes all of the power-up tests described in Chapter 14, but the tests are more extensive. It also provides testing of other generic functions of the MCP card. The second-level tests return the locations and descriptions of failures, described in Table 14-4.

MCP_Diagnostic may be set as the start-up application on a boot disk or started like any other application. This application is written in MPW Pascal, MPW C, and MPW Assembler. Samples included on the distribution disk are written in MPW Pascal; it is recommended that you use the file VendorBlocks.p written in MPW Pascal to interface your Assembler diagnostic routines to the MCP_Diagnostic application.

NuBus support

Several extensions are included with the basic MCP_Diagnostic routines to support the NuBus environment, including routines to

- Trap bus errors (caused by accessing nonexistent memory locations or by bad NuBus cards), provide information on the location of the problem, and return control to the application
- Provide general access to dialog boxes and dial controls
- Run scripted tests

MCP_Diagnostic main window

To best understand the user interface for this application, you should first run the MCP_Diagnostic application provided on the distribution disk and explore the various menus and commands. Each of these commands is described in the following sections of this chapter.

To run the second-level diagnostics for the MCP card:

1. Open the folder named MCP_Diagnostic.
2. Double-click on the application named MCP_Diagnostic. (The icon for this application is shown below.)



MCP_Diagnostic

Figure 15-1 shows the main window that is displayed.

MSC NNNN
ART: NN x 17 pi
20.5 pi text to FN b/b

Figure 15-1
MCP_Diagnostic main window

The main window contains oversized buttons to the left of the screen and two sets of bit rectangles located at the bottom of the screen.

Clicking the Start Test button runs through the entire suite of currently enabled diagnostic tests.

Clicking the Reset Test button stops and aborts the current test sequence.

Clicking the Pause button stops the diagnostic at the next possible break point and provides the message `Paused...Press Continue when Ready`. These break points are determined by a call to the routine `WaitStep()`, described later in this section.

MCP_Diagnostic

Start Test

Reset Test

Pause Test

Continue

ErrPause

ErrEnd

ErrLoop

ErrCont

31	28	24	20	16	15	12	8	4	0	
0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	\$00000000 actual
0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	\$00000000 expected

15-1

Clicking the Continue button starts the diagnostic (when paused) at the next test in the sequence.

The radio buttons located in the lower-left section of the window determine the action to be taken when a failure occurs during this sequence. The default is set to pause after a failure. You can change it by clicking on the buttons in the window, then saving using the Save Configuration option under the Options menu.

In a separate section on the main window, the Logger routine reports errors to the window and logs the errors to disk on the default drive.

The area at the bottom of the window is used to graphically show bit-level failures, which are displayed as actual bit rectangles versus expected bit rectangles. To set the expected bit levels, use the routine `ShowResults()`, described later in this chapter.

- ❖ *Note:* While running the current version of MCP_Diagnostic, it may take a while before operator input is acted upon, because user interface routines are called only between the 32-bit addressing tests. Some tests take up to five minutes to complete (such as the byte-inversion RAM test).

During RAM testing, you may see how this display is used to report bit errors, with the position of bit differences indicating which RAM chips failed.

The next section describes the commands presented by the MCP_Diagnostic application. You'll see the following menus when you run the application:

- MCP
- Options
- Debug Aids
- Display

The features and functions of each of these menus are described next. To determine the version date of the MCP_Diagnostics application, choose the "About MCP..." command under the Apple menu.

MCP menu

The MCP menu, shown in Figure 15-2, defines the card slots you want to test and the scripting controls. This menu is reserved for testing the MCP card.

- ❖ *Note:* Other expansion cards or smart cards, such as the AST card, are not recognized by MCP_Diagnostic.

MSC NNNN
ART: NN x 17 pi
20.5 pi text to FN b/b

Figure 15-2
MCP menu

Slot n

When you first run the application, MCP_Diagnostic automatically looks for MCP cards and adds a check mark beside any slot in which an MCP card is found. All available slots in the Macintosh II computer at startup are listed in this menu. For example, if an MCP card is installed in Slot D, MCP_Diagnostic adds a check mark to the Slot D menu item. MCP_Diagnostic determines that a card is an MCP card if the application finds the value \$C3D2 at the card ROM offset \$FFFFEA.

Before any sequential tests are actually executed, MCP_Diagnostic checks the slots that are marked under the MCP menu to verify that memory exists in the ROM address space. If there is no card installed or the MCP_Diagnostic finds neither ROM nor the MCP ROM identifier, the warning dialog box shown in Figure 15-3 is displayed.

MSC NNNN
ART: NN x 17 pi
20.5 pi text to FN b/b

Figure 15-3
Warning dialog box

If the board is not an MCP card, you can deselect it for testing by clicking the cancel button in the warning dialog box; to continue testing the selected cards, click OK.

Failure Analysis

Use the Failure Analysis command to show the number of failures and the number of iterations of each test after tests have been run. This information can also be found by choosing the Show Data Log command.

Run Script...

Use the Run Script... command to compile a specific script file and execute it one time only. This command presents a menu dialog box, shown in Figure 15-4.

```
MSC NNNN  
ART: NN x 17 pi  
20.5 pi text to FN b/b
```

Figure 15-4
Menu dialog box

Select the name of the desired script using the scroll bar. When invoked, MCP_Diagnostic loads and compiles the desired script. If MCP_Diagnostic detects compile errors, it displays messages in the scrolling text window. If compilation is successful, MCP_Diagnostic executes the script.

Run Script Repeatedly...

The Run Script Repeatedly... command allows you to choose how many times you want to execute the script you selected using the Run Script... command just described. You may enter a number or select uninterrupted repetition (that is, run the test until you turn off the computer) in the dialog box shown in Figure 15-5.

MCP_Diagnostic

Start Test

Reset Test

Pause Test

Continue

ErrPause

ErrEnd

ErrLoop

ErrCont

📁 MCP

Generate_MCP

MeasLog

TestLog

Vendor.a

Vendor.r

VendorBlocks.p

Vendordefs.p

New Baby

Eject

Drive

Open

Cancel

31	28	24	20	16	15	12	8	4	0	\$00000000	actual
										\$00000000	expected

15-4

MSC NNNN
ART: NN x 17 pi
20.5 pi text to FN b/b

Figure 15-5
Run Script Repeatedly dialog box

Run Script at Startup...

The Run Script at Startup... command allows you to select a script to be automatically executed when the MCP_Diagnostic is started. Execution begins after the test last completed.

Run Level Three Shell ...

Use the Run Level Three Shell... command to invoke third-level menus and tests. Refer to Chapter 16 for complete information on these diagnostics.

VendorMenu Item

Use the VendorMenu Item command to extend the MCP menu by adding menu commands you create. To add commands, use the files provided on the MCP Diagnostics distribution disk and described in Chapter 16. The scripting commands used with this menu item are also explained in detail in Chapter 16.

Options menu

The Options Menu, shown in Figure 15-6, is used to select test startup configuration, to quit the application, or to restart the Macintosh II.

MCP_Diagnostic

-
-
-
-

- ErrPause
- ErrEnd
- ErrLoop
- ErrCont

Set the number of script repetitions:
3

31	28	24	20	16	15	12	8	4	0	\$00000000	actual
										\$00000000	expected

15-5

MSC NNNN
ART: NN x 17 pi
20.5 pi text to FN b/b

Figure 15-6
Options menu

Auto Run is Selected

Use the Auto Run is Selected command to execute selected tests immediately at startup. Use the Save Configuration command to save this option.

Auto Run is NotSelected

Use the Auto Run is Not Selected command to disable the Auto Run function described above.

Save Configuration

Use the Save Configuration command to save the current configuration of the diagnostics. This command presents a dialog menu box similar to that shown in Figure 15-7 that asks for the name of a file where you want to save test data. The Default for this file is `Options.OPTN`.

MSC NNNN
ART: NN x 17 pi
20.5 pi text to FN b/b

Figure 15-7
Save Configuration dialog box

MCP_Diagnostic

Start Test

Reset Test

Pause Test

Continue

MCP_Diagnostics

- Build_Diag_Binary
- DiagLib.o
- DiagTool
- MCP Release Notes
- MCP_Diagnostic
- ROM

Save Test Data as:

Options.OPTN

New Baby

Eject

Drive

Save

Fig 15-7

Quit

Use the Quit command to exit to the desktop.

Eject and Reset

Use the Eject and Reset command to eject any disks in the Macintosh II and reboot the computer.

Debug Aids menu

The Debug Aids menu, shown in Figure 15-8, is used to control various options available while running the tests.

MSC NNNN
ART: NN x 17 pi
20.5 pi text to FN b/b

Figure 15-8
Debug Aids menu

Stop After Pass

Use the Stop After Pass command to run through the sequence of selected tests only once. This menu item is already checked; if you deselect this command, these tests will run continuously.

Enable Micro Stepping

Use the Enable Micro Stepping command to pause between the testing of each MCP card. When testing is paused, a MicroStep button on the left side of the main window appears. Every test is performed sequentially on each selected card before starting the next test in the sequence.

When you stop a test by clicking the MicroStep button and want to continue testing after a pause, click the MicroStep button on the left side of the main window.

Enable One Test Stepping

The Enable One Test Stepping command performs each test once on all MCP cards before pausing.

Enable Verbose Data Logging

The Enable Verbose Data Logging command stamps each test with the date and logs each test as it begins to test both the log file and the data log window (described later in this chapter). When this command is unchecked, only errors are logged; as a result, testing runs quite a bit faster.

Zero Data Log File

Use the Zero Data Log File command to clear the log file of information pertaining to previously run tests. A dialog box appears to confirm the zero data log, shown in Figure 15-9.

MSC NNNN
ART: NN x 17 pi
20.5 pi text to FN b/b

Figure 15-9
Zero Data Log File dialog box

MCP_Diagnostic

Start Test

Reset Test

Pause Test

Continue

ErrPause

ErrEnd

ErrLoop

ErrCont



Are you certain you want to
erase the data log?

OK

Cancel

31	28	24	20	16	15	12	8	4	0		
0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	\$0000000	actual
0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	\$0000000	expected

15-9

Clear Graph

The Clear Graph command is not implemented in this version.

Disable All Logging

Use the Disable All Logging command to run selected tests without reporting errors or comments to the disk file. This command is then dimmed (grayed) in the menu.

Display menu

The Display menu, shown in Figure 15-10, is used to display information about the tests being run and the resulting status.

MSC NNNN
ART: NN x 17 pi
20.5 pi text to FN b/b

Figure 15-10
Display menu

Show Controls

The Show Controls command displays the complete list of MCP card tests. The generic MCP tests provided on the MCP Diagnostic disk cover the following:

- RAM
- ROM
- 68000
- NuBus
- Interrupts

- Show Controls
- Show Bits
- Show Data Log
- Show Measurement Log
- Show Graph
- Ignore Show Bits

Start Test

Reset Test

Pause Test

Continue

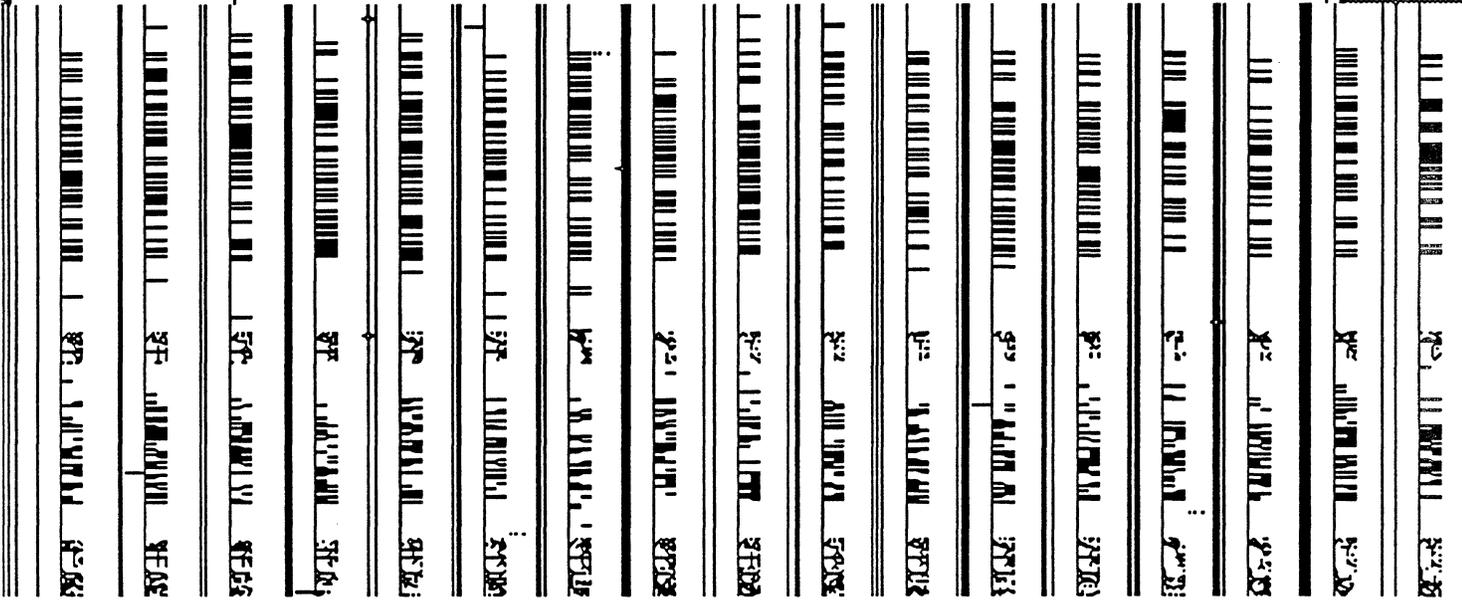


Fig 15-10

Related tests are usually grouped together in blocks (such as the RAM test block). You may use as many individual tests or blocks of tests as necessary to structure the tests for the smart card you create. You enable or disable tests by selecting check boxes in the list of tests for each test block, shown in the figures for each test block that follow.

- ❖ *Note:* Except for RAM testing, you need a current version of the MCP declaration ROM (Revision D) to successfully run through the test sequence.

Testing RAM on the MCP card

This series of tests comprehensively checks the card's available RAM for data and address errors. Figure 15-11 shows Block 1 — RAM testing.

MSC NNNN
 ART: NN x 17 pi
 20.5 pi text to FN b/b

Figure 15-11
 RAM test blocks

Table 15-1 describes the tests performed on RAM for the MCP card.

Table 15-1
 Tests on-card RAM

Test type	Description of test
Data Lines	Cycles all data values through one location
Stuck Cell	Zeros memory, and then sets
Address Line	Writes an incrementing pattern to memory and verifies
Fixed Patterns	Writes a set of ten 128-bit patterns to RAM and verifies
Marching Ones	Cycles bits through each RAM location
Byte Inversion	Tests the ability of RAM to invert data patterns in each RAM location

- ❖ *Note:* Byte Inversion is the longest of all tests, requiring about three minutes to complete for a 1/2 megabyte of RAM.

MCP_Diagnostic

Start Test

Reset Test

Pause Test

Continue

ErrPause

ErrEnd

ErrLoop

ErrCont

Block & Test Switches

- Block 1 - RAM Testing
- Test 1 - Check Data Lines
- Test 2 - Stuck Cell Test
- Test 3 - Check RAM Address Lines
- Test 4 - Fixed Patterns
- Test 5 - Forward Marching 1's
- Test 6 - Reverse Marching 1's
- Test 7 - Byte Inversion

15-11

Bit errors are reported in the section of the screen that displays the expected bit rectangles versus actual bit rectangles, located at the bottom of the main window.

Table 15-2 shows how errors are returned for each of the tests just listed.

Table 15-2
How errors are returned for RAM tests

Test type	How errors are returned
Data Lines	Returns bit failures in lower portion of main window
Stuck Cell	Accumulates and displays error bits in the main window
Address Line	Reports address and bit field discrepancies on an error
Fixed Patterns	Returns bit field discrepancies on an error
Marching Ones	Returns both error address and value discrepancies in the main window
Byte Inversion	Sets and inverts each byte, reporting errors if any bit does not set or clear

❖ *Note:* Power-on and primary initialization tests indicate what failed by the error code shown in Table 14-4. Because of dependencies in the diagnostic application, if power-on or primary initialization tests fail, spurious error messages may occur for remaining tests.

To avoid losing the initial board information created by the power-up tests, the memory in the first 512 bytes of RAM is saved before and restored after the tests.

❖ *Note:* Since each board uses a different labeling system for its RAM, you must either replace all RAM chips or consult the hardware specifications for identifying a faulty chip.

Testing ROM on the MCP card

The test constructs the Cyclic Redundancy Check (CRC) word for the ROM, and tests this word against what is found in the card's information block located near the top of ROM space.

Figure 15-12 shows Block 2 — ROM testing.

MCP_Diagnostic

Start Test

Reset Test

Pause Test

Continue

ErrPause

ErrEnd

ErrLoop

ErrCont

Block & Test Switches

Block 2 - ROM Testing

Test 1 - CRC Test

Test 2 - Slot Manager Verification

Fig 15-12

MSC NNNN
ART: NN x 17 pi
20.5 pi text to FN b/b

Figure 15-12
ROM test blocks

If this test fails, it returns a number; refer to the information on Status Results for the Slot Manager in *Inside Macintosh*, Volume 5. More information about this test can be found in the power-up source code on the MCP_Diagnostic distribution disk.

Testing the 68000

The Macintosh II triggers the initial power-up and primary initialization tests. Figure 15-13 shows Block 3 — 68000 testing.

MSC NNNN
ART: NN x 17 pi
20.5 pi text to FN b/b

Figure 15-13
68000 test blocks

The results are placed in the RAM of the MCP card and control is returned to the 68020 on the Macintosh II to interpret the returned status.

MCP_Diagnostic

Start Test

Reset Test

Pause Test

Continue

ErrPause

ErrEnd

ErrLoop

ErrCont

Block & Test Switches

Block 3 - 68000 Testing

Test 1 - Execute Power-On Tests

Test 2 - Execute Primary Initialization Tests

Fig 15-13

The ROM power-up status failure can be stored in long word location \$134 (hex). The application calls the routine `VendorPowerUpStatus()` in the file `Vendorblocks.p` to interpret the failure of the Vendor Power-up Test. You can store the primary initialization status failure in long word location \$138. Use the routine `VendorPrimaryInitStatus()` to interpret the failure of the Vendor Primary Init Status test.

- ❖ *Note:* Power-on and primary initialization tests indicate what failed by the error code shown in Table 14-4. Because of dependencies in the diagnostic application, if power-on or primary initialization tests fail, spurious error messages may occur for remaining tests.

Testing NuBus

Nubus tests can be performed when NuBus is used in the following operations:

- reading from Macintosh II system ROM
- writing to Macintosh II system RAM
- reading from Macintosh II system RAM
- using MCP interprocessor tests
- test and set using 68000 memory
- test and set using 68020 memory

Figure 15-14 shows Block 4 — NuBus testing.

MSC NNNN
ART: NN x 17 pi
20.5 pi text to FN b/b

Figure 15-14
NuBus test blocks

- ❖ *Note:* Power-on and primary initialization tests indicate what failed by the error code shown in Table 14-4. Because of dependencies in the diagnostic application, if power-on or primary initialization tests fail, spurious error messages may occur from the diagnostics for NuBus tests.

MCP_Diagnostic

Start Test

Reset Test

Pause Test

Continue

ErrPause

ErrEnd

ErrLoop

ErrCont

Block & Test Switches

Block 4 - NuBus Tests

Test 1 - Read from Mac II system ROM

Test 2 - Read from Mac II system RAM

Test 3 - Write to Mac II system RAM

Test 4 - 32 Bit Mac II system RAM R/W

Test 5 - Write/Read to Caliente card RAM

Test 6 - Test & Set on Caliente RAM

Test 7 - Test & Set on Mac II system RAM

Test 8 - Caliente-Caliente InterProcessor

15-14

Reading from Macintosh II system ROM

The MCP Sequencer reads and verifies a specific value from the ROM address \$800008 (Macintosh ID) in the System ROM that has a value of \$01.

Writing to Macintosh II system RAM

There are two versions of this test; you select the version using the Show Controls command in the Display menu.

- The first version writes a byte to the application scratchpad and reads it back. If the byte readback is the same as the byte that was sent, the test succeeds.
- The second version of the test verifies that a long value (four bytes) is written and read correctly.

Reading from Macintosh II system RAM

The MCP_Diagnostic checks the address \$000000 for a value of \$40 in the System RAM.

Warning

If this test fails, it may be because some other application or the MultiFinder has changed the location. Try running this application without using MultiFinder.

MCP interprocessor tests

This test performs reads and writes from an MCP card to any other MCP card or cards that are installed on the Macintosh II main logic board (motherboard). If only one MCP card is installed on the Macintosh II main logic board, this test exits without error.

Test and set using 68000 memory

This test examines the ability of the two processors (one on the MCP card, the other on the Macintosh II main logic board) to gain access to the same memory block. The card and the main logic board processor attempt many test-and-sets to the same memory location. This test fails if both processors show that they have gained access to the location for any one test cycle.

Test and set using 68020 memory

This test is functionally equivalent to the one just described, except that the memory block resides in 68020 system memory.

Reset/timer/interrupts

The MCP Sequencer performs several tests to check the reset, timer, and interrupts performance of the MCP card and includes the following:

- Level 1 timer interrupt
- Level 1 timer speed verification
- Level 2 NuBus interrupt
- the MCP card interrupts the Macintosh II 68020

Figure 15-15 shows Block 5 — Interrupt testing.

MSC NNNN
ART: NN x 17 pi
20.5 pi text to FN b/b

Figure 15-15 Interrupt test blocks

- ❖ *Note:* Power-on and primary initialization tests indicate what failed by the error code shown in Table 14-4. Because of dependencies in the diagnostic application, if power-on or primary initialization tests fail, spurious error messages may occur from the diagnostics for NuBus tests.

Level 1 timer interrupt

This test performs checks if timer interrupts are processed by the card. A test routine is loaded into the card. Its address is placed in the timer interrupt slot in the card's exception table, and a test is triggered. If the routine executes, the test passes.

Level 1 timer speed verification

This test checks the average speed of the 68000 timer interrupt, normally firing once every 6.5536 milliseconds. If this interrupt is much faster or slower than the specification, the failure is reported to the main window and recorded to disk.

MCP_Diagnostic

Start Test

Reset Test

Pause Test

Continue

ErrPause

ErrEnd

ErrLoop

ErrCont

Block & Test Switches

Block 5 - Interrupt Testing

Test 1 - Timer Interrupt

Test 2 - Timer Interrupt Speed Verification

Test 3 - NuBus Interrupt

Test 4 - I/O Processor Interrupt

15-15

Level 2 NuBus Interrupt

This tests the ability of the 68020 to interrupt the card's 68000 processor. A test routine is downloaded to the card and triggered. This test passes if NuBus interrupts are correctly processed by the card.

The MCP Card interrupts the Macintosh II 68020

This test uses the NuBus NMRQ line to trigger a slot interrupt, which is then processed by the Device Manager on the Macintosh II. If the interrupt is triggered, the test passes.

Show Bits

Use the Show Bits command in the Display menu to update the bit rectangles at the bottom of the main window. Currently, only the RAM tests use the bit rectangles to show which RAM data bits failed.

Show Data Log

The Show Data Log command displays all errors that have occurred since the Zero Data Log command was last used. These errors are displayed in the Data Log window to allow the user to view the data log within the MCP_Diagnostic. Figure 15-16 shows an example of how a Data Log window displays information after tests have been run.

```
MSC NNNN  
ART: NN x 17 pi  
20.5 pi text to FN b/b
```

Figure 15-16
The Data Log window

If you have not yet run any tests and then click on Show Data Log, a dialog box is displayed that tells you Data Log File is Empty. Click OK.

Data Log

Updated: 10/25/88 3:45 PM Started: 10/18/88 4:24 PM
Iterations Since Last Zero Data Log: 2, Errors Since Last Zero Data Log: 0

Failed Slot/Block/Test/# of Errors Since Bootup

10/18/88, 4:25 PM, Pass:0, Bk:1, Test:1, Slot:B: test 1(begun test)
10/18/88, 4:25 PM, Pass:0, Bk:1, Test:2, Slot:B: test 2(begun test)
10/18/88, 4:25 PM, Pass:0, Bk:1, Test:3, Slot:B: test 3(begun test)
10/18/88, 4:25 PM, Pass:0, Bk:1, Test:4, Slot:B: test 4(begun test)
10/18/88, 4:25 PM, Pass:0, Bk:1, Test:5, Slot:B: test 5(begun test)
10/18/88, 4:25 PM, Pass:0, Bk:1, Test:6, Slot:B: test 6(begun test)
10/18/88, 4:25 PM, Pass:0, Bk:1, Test:7, Slot:B: test 7(begun test)
10/18/88, 4:27 PM, Pass:0, Bk:2, Test:1, Slot:B: CRC test(begun test)
10/18/88, 4:27 PM, Pass:0, Bk:3, Test:1, Slot:B: (power up)(begun test)
10/18/88, 4:27 PM, Pass:0, Bk:3, Test:2, Slot:B: Primary init test(begun test)
10/18/88, 4:27 PM, Pass:0, Bk:4, Test:1, Slot:B: Mac II ROM Read(begun test)
10/18/88, 4:27 PM, Pass:0, Bk:4, Test:2, Slot:B: Mac II RAM Read(begun test)
10/18/88, 4:27 PM, Pass:0, Bk:4, Test:3, Slot:B: Mac II RAM write(begun test)
10/18/88, 4:27 PM, Pass:0, Bk:4, Test:4, Slot:B: Long Mac II RAM write(begun test)
10/18/88, 4:27 PM, Pass:0, Bk:4, Test:5, Slot:B: Native Units Read test(begun test)

15-16

Show Measurement Log

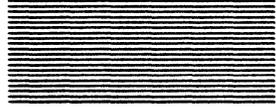
The Show Measurement Log command is not implemented in this version.

Show Graph

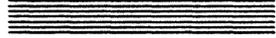
The Show Graph command displays an empty graph; this command is not implemented in this version.

Ignore Show Bits

The Ignore Show Bits command is not implemented in this version.



Chapter 16



Adding to MCP

This chapter describes how to add resources and application-specific tests to the ROM on MCP-based smart cards as well as menu commands to the MCP_Diagnostic application using routines provided on the MCP Diagnostics distribution disk. The sections in this chapter pertain to sequential tests only; see Chapter 17 for information on coprocessor testing.

The MCP_Diagnostic application is written in MPW Pascal, MPW C, and MPW Assembler. Therefore, any tests you create should also be written using MPW. This section assumes you have working knowledge of MPW Pascal, MPW C, MPW Assembler, or all three.

- ❖ *Note:* While it is possible to use C for writing tests, you must exercise care in string conversions between Pascal, toolbox calls, and C. All toolbox calls from C must be declared `pascal () extern`, as the Pascal libraries are used first during the link phase. Toolbox calls requiring strings must pass Pascal strings.

Adding code to the ROM

The MCP_Diagnostic application provides hooks and a user interface that you can use to customize the files; this code is provided in the MCP folder on the MCP Diagnostics distribution disk.

You must write your own code for the parameters and device drivers for your specific card application. Hooks are set up for them in the ROM's resource directories. Refer to *Designing Cards and Drivers* for more information.

To add your own code to the ROM on the MCP card, be aware that

- The Slot Manager requires that resources on the ROM be created as specified in *Designing Cards and Drivers*.
- Both the generic power-on code and the primary-initialization code have hooks where you may add code for your own tests.

The MCP Diagnostics distribution disk provides a set of folders and stub files that are set up to make development easier. The ROM files are in the folder named ROM. Inside the folder are several files and a folder named MCP.

Table 16-1 provides a list of the files in the MCP folder and a brief description of each. Each of these files is described in greater detail in the following sections in this chapter.

Table 16-1
ROM:MCP files

Name	Description
App1ROM.a	Defines the slot resource tables
App1PowerOn.a	Used to include your tests into the power-on sequence
App1PrimaryInit.a	Used to add tests into the primary initialization sequence
Application.h	Contains the constants required for your code
ROM burn instructions	Contains information for downloading and burning your code into ROM

Warning

Do not make any changes to the files PowerOn.a or PrimaryInit.a contained in the ROM folder, because these files may be replaced or modified in the future.

Duplicate the :ROM:MCP: folder and rename it for your project. The name of this new folder should not contain any spaces or special characters, as it is used to locate your code for the ROM makefile.

The file App1ROM.a

The file App1ROM.a contains slot resource lists, listed in Table 16-2. For a more complete description of MCP card resources, consult *Designing Cards and Drivers*.

Table 16-2
Resource list entries in the file App1ROM.a

Name	Description
Board sResource list	Holds pointers to the MCP card resources, as well as the vendor identification
CPU sResource list	Holds information describing the CPU and memory map of the MCP card architecture
Application drivers sResource list	Holds any drivers that will be loaded at startup; also holds the category and type information for your card's specific use

❖ *Note:* Consult Apple Developer Services for the latest values for categories and types.

Board sResource list

The resource list holds pointers to the MCP card resources `sRsrc_Board` and `VendorInfo`.

The `sRsrc_Board` list contains the name and identification number for the MCP card. The parameters include:

- The `_BoardName`, the official card name you designate
- `CommBoardID`, the Board ID designated for the application by Apple Developer Technical Support

Important

You must apply to Apple Developer Technical Support for an official Board ID. Once this Board ID is received, change the constant (declared in the file `Application.h`) to the given value.

The `VendorInfo` record contains three fields. These fields are optional to the Slot Manager.

- `_VendorID`, the name of the developing company
- ❖ *Note:* You may want to include a copyright statement in this field, which is limited to 254 characters.
- `_RevLevel`, the current revision number you designate (less than 9)
- `_PartNum`, the part number you designate (according to your numbering system)

Application-specific driver sResource list

The driver directory is used to hold any drivers that will be loaded at startup. Refer to the sample code in Chapter 8 of *Designing Cards and Drivers*.

`VendID` resides immediately before the Format Header Block and is used by `MCP_Diagnostics` routines to recognize the card's specific application.

- ❖ *Note:* `VendID` is defined in the file `Application.h`. Until such time as Apple Technical Support supplies you with a Board ID number, you should define your own code and connect this into the second level of diagnostics, using the function `IsVendorCard()` in file `VendorBlocks.p`. If `IsVendorCard` returns false, your application-specific tests will not be executed.

The file `AppIPowerOn.a`

The file `AppIPowerOn.a` is used for two purposes. The first is to include tests in the power-on or soft reset code, which is run by the on-board 68000. Do not put any time-intensive tests here, as any tests you develop must not take more than 600 milliseconds to execute (in a worst-case scenario).

The second purpose is to define the routines to service the interrupt vectors. This file allows you to add your own routines.

The power-on code recognizes a Pass or Fail status when the test bit flag (bit 4 at card location \$102) is cleared or set. The current stub code clears the bit, indicating a Pass. For failure, leave the bit set. A long word in card RAM at location \$134 is reserved for storing the failure status of any developer-created tests for later analysis.

The file `AppIPrimaryInit.a`

The file `AppIPrimaryInit.a` is used to include your test in the primary initialization code, which is then run by the Slot Manager. Tests placed here should take less than 200 milliseconds to perform.

If the card must be initialized, it should be done during primary initialization. The card should be left in a known state in which it is ready to be used by an application. A long word at \$138 has been reserved to store any test failure codes for later analysis.

The file `Application.h`

The file `Application.h` contains many constants required for your code. Currently, the constants provided by Apple Developer Technical Support for slot resources are defined in this file; these constants are used in the file `AppleRom.a`.

The file `ROMburn`

This file provides the information necessary to take the ROM application file and download it to a data I/O PROM burner.

Adding required resources in the ROM

This section describes a resource list called the sMemory resource that is provided in ROM on smart NuBus cards capable of being NuBus masters. A smart card for purposes of this discussion is any card with a CPU capable of being a NuBus master and providing memory-like access to its RAM.

- ❖ *Note:* An example of an implementation of the sMemory resource list can be found in the file `App1ROM.a` in the ROM folder on the MCP Diagnostic distribution disk.

This resource list provides information on the address ranges on MCP-based cards used for RAM, ROM, and/or device registers. This information is used by MR-DOS software to allow intelligent cards in the NuBus to communicate as peers. The Macintosh Coprocessor Platform and cards built upon this architecture implement this resource list.

- ❖ *Note:* It is strongly recommended that the NuBus interfaces on intelligent cards fully support all modes of NuBus access, even if multiple local bus cycles are required to complete them. Supplying this resource list on cards that do not meet all of these criteria is also acceptable.

sMemory resource list

The sMemory slot resource list is currently defined as a second-level slot resource list as part of the first level CPU slot resource list. The implication is that they are not visible to the the primary Slot Manager calls `SNextSRsrc()` and `SNextTypeSRsrc()`, but are accessible by use of the advanced level Slot Manager toolbox calls.

The identifier for the sMemory resource list is the Apple-reserved ID \$6C.

In MCP-based cards, these resource lists are located as an offset from the CPU resource list. Figure 16-1 illustrates what could be the sMemory resource list for a generic MCP card.

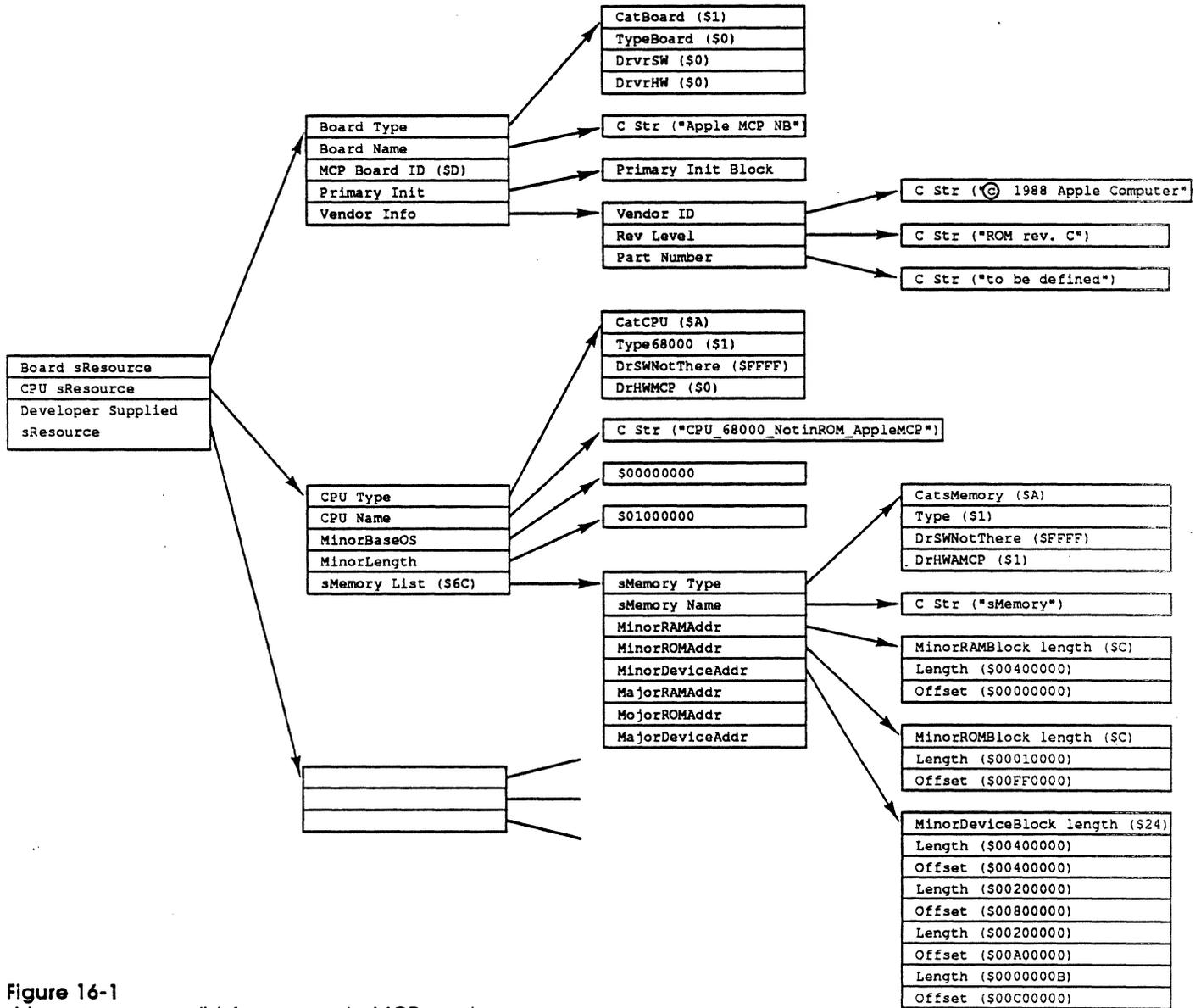


Figure 16-1
sMemory resource list for a generic MCP card

Adding required resources in the ROM 16-7

sMemory resource list identifier

The sMemory resource list contains the required `sRsrc_Type` (that identifies the list), the required `sRsrc_Name` resource, and one or more of the following resources listed in Table 16-3.

Table 16-3
Resource list for smart cards

ID	Description
128	Minor RAM address ranges
129	Major RAM address ranges (not supported on the MCP card)
130	Minor ROM address ranges
131	Major ROM address ranges (not supported on the MCP card)
132	Minor device register address ranges
133	Major device register address ranges (not supported on the MCP card)

- ❖ *Note:* A major address range corresponds to what is defined in *Designing Cards and Drivers* as the super slot space allocated for the card; however, the MCP card does not support the super slot space. A minor address range is the normal slot 16 MB space (such as where an MCP card resides) and is supported.

In each case, the upper 8 bits holds the ID and the lower 24 bits provides an offset to a block. The first long word of the block contains the length of the block, followed by pairs of entries. The first long word of each pair has the length of this address range in bytes. The second long word has the offset from the major or minor base address for this space, as appropriate.

It is acceptable for the resource list to describe the architected memory structure of the card; it need not reflect the actual memory present. For example, if 512 KB of RAM is provided for but only 128 KB of RAM is present, it is acceptable for the resource to indicate 512 KB of RAM space even though the remaining 384 KB of address space may either mirror the first 128 KB or cause a bus error when accessed. This means that the declaration ROM need not change when memory is expanded.

The card must return either a bus error or a data acknowledgement for any memory access within the architected memory range.

Table 16-4 lists the resources required for use with MR-DOS on an MCP-based smart card. This table includes the resource identifier and a brief description of each.

Table 16-4
Resources required for MR-DOS

ID	Resource Name	Description
1	sMemory Type	points to the eight-byte sRsrc_Type that identifies this list
2	sMemory Name	points to the required sRsrc_Name string by convention, sMemory
128	MinorRAMAddr	points to the range list for minor RAM space. In this case, up to 4 MB of RAM starting at 0 from the minor base address
130	MinorROMAddr	points to the range list for minor ROM space. In this case, 64 KB of ROM starting at \$FF0000 from the minor base address
132	MinorDeviceAddr	points to the range list for device space. In this case: 4 MB of space for I/O interface logic (decode and DTACK provided) starting at \$400000 from the minor base address 4 MB of space for I/O interface logic (no decode or DTACK provided) starting at \$800000 from the minor base address 2 MB of space for NuBus addressing starting at \$A00000 from the minor base address 12 bytes of space for MCP control addresses starting at \$C00000 from the minor base address

Source files for adding tests

MCP_Diagnostic routines are provided as object files compatible with the MPW 2.0 linker. Table 16-5 lists the MCP_Diagnostic files available for you to use in adding application-specific tests and provides a brief description of each.

Table 16-5
MCP_Diagnostic routine files

File Name	Description
Generate_MCP	Includes a sample compile and link format that can be used to generate a working version of the MCP_Diagnostic. Any additional developer files created for testing should be included in this existing MPW script file
VendorDefs.p	Is used for defining developer-specific constants, variables, and functions that are to be used globally (that is, in more than one file)
VendorBlocks.p	Contains stubs of the test routines that implement your Pascal application-specific tests on the MCP card. Tests needing to access address space above the 1-MB card space are written in Assembler (this is explained in a later section)
Vendor.a	Used for creating application-specific Assembler routines
Vendor.r	Contains vendor-specific resources necessary to the vendor-defined tests. This file includes the names of the tests to be run, along with any dialog or alert boxes for these same tests

You should create new tests as separate files and link them together using the model found in the file `Generate_MCP`.

Including new tests in the MCP_Diagnostic

Tests you create become part of the MCP_Diagnostic by

- including the names of the tests to be added into the file `Vendor.r`
- initializing the test routines using `MkNewBlock()` and `MakeNewTest()` in the file `VendorBlocks.p`
- calling the test routines from the routine `VendorExecTest()` in the file `VendorBlocks.p`

The stub tests in the file `VendorBlocks.p` provide an example of how this is done.

Code that must run before each start of a test sequence should be placed in the routine `VendorStartTest()`, and code to be run after each test sequence is complete should reside in the routine `VendorEndTest()`. Any code to be executed once upon startup of the application should be located in the routine `VendorStartup()`, and code to be run right before the close of the application should reside in the routine `VendorExit()`. Stubs to these routines are also found in the file `VendorBlocks.p`.

The tests log errors and present information to the operator via the `MCP_Diagnostic` interface routines. Each test should be (or look as if it is) a Pascal function returning a 16-bit integer indicating success (zero value) or failure (non-zero error status value). An example is provided in the routine `VendorExecTest()` in the file `VendorBlocks.p`.

In order for `MCP_Diagnostic` to recognize different board applications built on the MCP card, you must use the routine `IsVendorCard(slot#)` to return `TRUE` if the slot contains your application board; otherwise, it should return `FALSE`. This routine is used to prevent `MCP_Diagnostic` from running the your application-specific tests on generic MCP cards. This stub resides in the file `VendorBlocks.p`.

Adding menu commands to the MCP_Diagnostic

You can add menu commands by extending the MCP menu. This is accomplished by

- extending the menu resource found in the file `Vendor.r`
- putting the code that supports these commands into the routine `VendorMenu()` in the file `VendorBlocks.p`

❖ *Note:* The parameters for all vendor routines are described in the file `VendorBlocks.p`.

Macintosh address mode compatibility

To remain compatible with Macintosh 68000 applications, the 68020 Macintosh II normally runs in 24-bit mode.

In 24-bit mode, only the first megabyte of each card's slot space can be seen by the application. Because of this, the MCP's control space and ROM cannot be accessed in 24-bit mode.

In order to access this space, you must call the Macintosh II ROM trap `_swapmmu` with the value not zero in register D0 to switch to 32-bit mode. As the `MCP_Diagnostic` user interface runs in 24-bit mode, the tests must reset the 24-bit mode upon completion of any 32-bit mode testing.

Trapping bus errors

When you access the NuBus address space on the Macintosh II, bus errors frequently develop if you address a nonexistent or nonresponding memory address. The routines `InstallMyErrV`, `InstallOldV`, and `BusErrDialog` are provided on the `MCP_Diagnostic` library to automatically trap this error and provide information about the cause of this error.

The routine returns the address of the offending location and the code location that called it; this information is helpful in determining the cause of the bus error. Usually the error is caused by specifying a NuBus address that does not contain an MCP card or does not have ROM installed. The location reporting the error is of the form `F n XXXXXXXX`, where n is the slot number of the bus error, and `XXXXXXXX` is the offset into the card's memory map.

Currently, the diagnostic returns to the main menu, but future versions may exit the application at this point because the program stack contains data left over from the state of the machine before the bus error occurred.

The Dial routine

The `Dial` routine facilitates receiving information between you and the diagnostic. This routine is used to "dial" a value for a variable. Use the command `RunScript Repeatedly...` under the MCP menu to bring up the window for the dialog.

The `Dial` routine (`Function SetVar`) found in the `MCP_Diagnostic` library returns an integer corresponding to one of three buttons selected in the dialog box. The format for the `Dial` routine is:

```
Function SetVar(var current:Integer; min,max:Integer;  
               msg1,bPrompt:str255): Integer;
```

where `msg1` is a prompt describing the actions of the dial (`msg1` should be less than 48 characters in length).

The variable `current` passes in the default location of the scroll bar indicator and returns the indicator controls value, which is between minimum and maximum in value.

If `buttonMsg` is not an empty string, a button appears with this string inside and the function returns -1 if clicked. The OK button returns 0, and the Cancel button returns +1 if pressed.

The tester script language

The MCP card testing application has the capability of being driven by a script file. The file you create must be an ASCII text file, which you can generate in a text editor. The file consists of two parts, described in the following sections:

- a section that contains script control statements
 - a section that contains an optional set of messages
- ❖ *Note:* The message section must follow the control section.

The control section

The control section consists of test descriptions, conditional statements, labels, and comments. As these are delimited by special characters, you can spread any element over several text lines; the script is case-sensitive. The control section is terminated by an `END` statement.

The testing application lets you build sets of tests called blocks. Each block is given a two-digit number; individual tests within the block are also identified by two-digit numbers. The block and test numbers form the TestID.

Using this ID, form a test description using the following format:

```
TestID [-q [e] [u] [t 14:10] [o] [n #]] [-s n ...] [-m #] *
```

Table 16-6 describes each of the parameters in the format in the test description.

Table 16-6
Test description parameters

Parameter	Description
Test ID	a four-digit number. The first two digits are the block number; the second two digits are the test number
-	termination mode. The default is to run the test once. A test runs until its termination mode occurs, and returns a failed status if it fails at least once
-qe	terminate on error
-qu	user terminate
-qo	run test once
-qn	run test <i>n</i> number of times
-qt <i>xx:xx</i>	termination time in 24-hour mode
-s	card slot(s). This is followed by one or more slot numbers in the range 9 - E. If no number is specified, the slot that was chosen from the menu is used. If no slot has been chosen, a dialog box prompts the user for a slot number.
-m	the number of a message that is printed on the screen if the test fails. The message must be present in the message section of the script.
*	the terminator character

Conditional tests

The script language has a limited set of conditional tests. These tests can be strung together to form more complex conditions.

Unlike most conditional tests, the conditional branch is only taken if the invoked test fails. If the invoked test passes, the flow of control jumps to the next distinct test description in order. Thus, if a test description consists of three chained conditionals and the first one passes, the second and third tests will be skipped.

The format for conditional tests is as follows:

IF (TestID) THEN (TestID) * If the given test fails, run the following test.

IF (TestID) GOTO label *

...

label

If the given test fails, jump to the label and continue. The label must stand alone on the line preceding the destination test and is limited to ten characters. The colon is required but is not counted as one of the ten. The first character of the label must be non-numeric.

TestID GOTO label *

Unconditionally jump to the label. This case is distinguished from the conditional GOTO by the fact that the TestID is not within parentheses and that the line does not start with an IF.

IF (TestID) THEN IF (TestID) *

String together a set of conditionals; the conditional must be terminated by an asterisk (*).

Examples

A complete example of a conditional test is

```
IF (0102 -qt 14:20 -s 9 A B) THEN (0203 -qo -s 9) *
```

This example shows that if test 2 of block 1 fails (which terminates at 14:20 and checks slots 9, A, and B), then execute test 3 of block 2 once on slot 9.

To create an IF ... THEN ... ELSE construction, use the following format:

```
IF (0102) GOTO test1*
```

```
0205 .... GOTO test2 *
```

```
test1:
```

```
0304 *
```

```
test2:
```

```
0109 ... *
```

This example shows that, if test 0102 fails, then execution continues with test 0304. If test 0102 passes, then test 0205 is executed. Finally, test 0109 is run.

The message section

The optional message section consists of a set of messages that can be printed when a test fails. Only one one-line message can be printed for each script line. Messages are numbered from 1 to 256; all messages are delimited by the symbol /, as shown in the following example:

```
/RAM test failed/  
  
/This message is about the maximum size for the message dialog box/  
  
/This message is illegal  
because it is on two lines/
```

Comments

Comments may be placed at any point in the control section, and are started by a semicolon. Any characters following a semicolon up to the end of the text line are ignored by the compiler. For multiline comments, each line must be preceded by a semicolon. Comments must not be inserted inside of messages.

Error reporting

You are informed if the compiler encounters any errors. The MCP Sequencer opens a scrolling text window and displays messages in it. When processing ends, the window is highlighted. The user may then scroll through the messages or exit by clicking the close box. Messages have the format

```
Offending line  
                ?  
Error message
```

The question mark points to the error point.

The compiler is a two-pass compiler followed by a link phase. The first pass checks labels for validity and builds a symbol table. The second pass builds the script table. Because of this, the same line of script may engender two messages at different points.

You may have the errors send to a text file on disk. To do so, include the command #DISK at the beginning of the script file. When the compiler sees this command, it opens the standard save file dialog box for you to specify the destination directory and file name. If you cancel the operation, the disk save is canceled.

Reserved words

The reserved words of the script language are

```
IF
THEN
GOTO
END
#DISK
```

The reserved characters are

```
(, ), :, /, :, -, #, and *.
```

The temporary file

When the script starts its execution, it opens a temporary file on disk. As each test is started, the `Test ID` is saved in this file. When the test has completed, the results of the test are appended to the file. When the script has finished, or if the operator cancels the script, the temporary file is deleted. In the case of power loss or system reset, the file will be left on the disk.

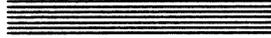
In the current version of `MCP_Diagnostic`, you cannot activate the temporary file.

Script control

The testing application has several menu items. Each of these are described in the section on the MCP Menus in Chapter 15.



Chapter 17



MCP Coprocessor Diagnostics

This chapter describes the third level of MCP Diagnostics. These tests allow the 68000 processor on the MCP to run concurrently with other 68000-based MCP cards and the 68020 on the Macintosh II. This chapter tells you how to start the coprocessor diagnostic tests, and provides additional information you need to customize an application for the board you create (based on the MCP card).

Warning

Some operator commands for customizing your own diagnostics will change in future versions. Specifically, the `GetMem()` and `FreeMem()` routines may not be supported in future MR-DOS compatible versions of the Level 3 shell. Future versions may use MR-DOS memory routines to provide the same functionality.

What are coprocessor diagnostics?

Coprocessor diagnostics are standalone programs written to run on an MCP-based card. Coprocessor diagnostics are controlled by the `MCP_Diagnostic` application, which provides a command language from a dedicated window for each NuBus slot containing an MCP-based card. The `MCP_Diagnostic` application provides communication and logging facilities for each card.

The next sections describe the third-level window and commands, as well as how to start coprocessor tests.

Entering third-level tests

To enter coprocessor tests, select `Start Level Three Shell...` from the MCP menu in the main window. The `MCP_Diagnostic` application creates a new screen and menu bar for third-level tests, and then displays a dialog box that allows you to select the destination directory for log files, as shown in Figure 17-1.

MSC NNNN
ART: NN x 17 pi
20.5 pi text to FN b/b

Figure 17-1
Third-level menu and dialog box

Select the desired directory and click Save. Clicking the Cancel button disables logging to a file.

The MCP_Diagnostic application next queries each slot to determine which slots contain valid MCP cards. For every valid card, the MCP_Diagnostic application creates

- an active slot table entry
- a window into a text file (shown in Figure 17-2)
- an associated log output file, which you can optionally cancel

MCP

- Generate...MCP
- MCP...Diagnostic
- Samprez.crd
- ScriptName
- Pendor.a
- Pendor.J

New Baby

Select logging volume:

27-1

MSC NNNN
ART: NN x 17 pi
20.5 pi text to FN b/b

Figure 17-2
Window into a text file

When you click in the desired window, that window and the slot represented become the active slot. The third level of the MCP_Diagnostic application can accept commands only from a window that is active. The output from coprocessor diagnostics is reported to each window, regardless of whether the window for that test is active.

Starting third-level tests

To start a third-level coprocessor application, follow these steps:

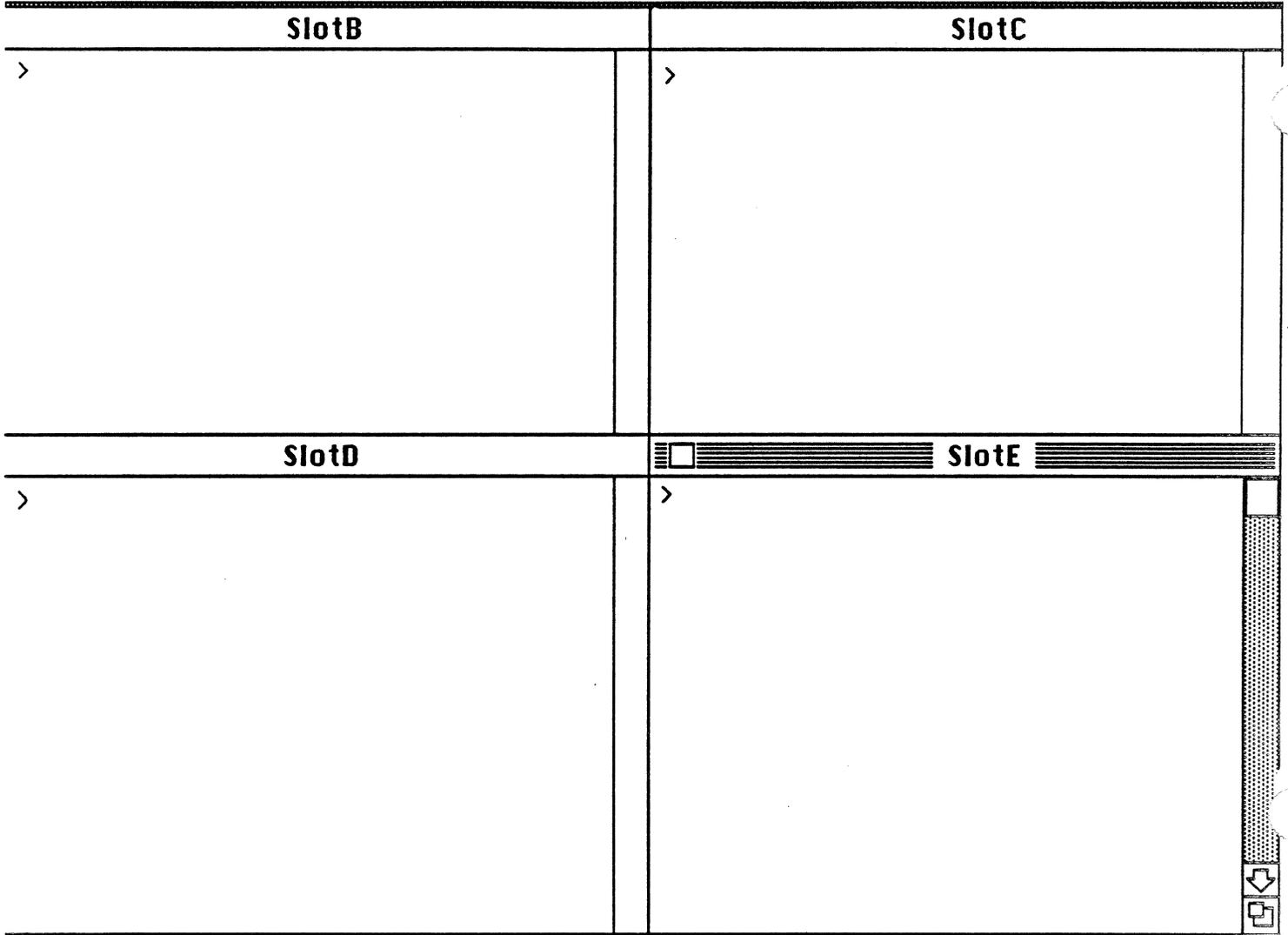
1. Move the cursor into the window representing the appropriate slot.
2. Click the mouse button.
3. Place the cursor after the last unoccupied caret.
4. Enter `Run < >`. Include the name of the file you created.

As a result of the command entered, MCP_Diagnostic application downloads a third-level coprocessor program or MR-DOS diagnostic program to the specified card and begins program execution on the card.

For example, enter

```
Run <echo>
```

This runs the sample level three task on the card to which the window belongs.



ms

CS

Third-level menus

The menu commands for third-level coprocessor diagnostics let you use this window as a mini text editor. There are a number of limitations, in that the text buffer can not exceed 32 KB. If you attempt to load in a file that exceeds this memory capability, only the last part of the file will be loaded.

You cannot use the Open command to open a window, but may use it to load a file into an already existing window. However, you may use the Close command to close the window.

File

The File commands operate on the window currently active. These commands allow you to enter the basic text editing commands, and operate in the standard Macintosh interface. For more information on each of these commands, refer to your owner's guide for the Macintosh II computer. The File menu is shown in Figure 17-3.

File

Open...

Close

Save

Save as...

Print Setup

Print

Quit

Figure 17-3
File menu for third-level tests

Quit

If logging was disabled upon entry into the third-level tests and messages have been sent to the windows, you are asked if you want to save the window contents. If you do, then the standard document Save window is shown for each window with text in it.

Edit

The Edit commands operate on the window currently active. These commands allow you to enter the basic text editing commands and operate in the standard Macintosh interface. The Edit menu is shown in Figure 17-4.

Edit

Cut/⌘	⌘H
Copy/C	⌘C
Paste/U	⌘U
Clear	
Select All/A	⌘A

Show Clipboard	

Figure 17-4
Edit menu for third-level tests

For more information on each of these commands, refer to the *Macintosh II Owner's Guide*.

MCP

The commands in the MCP menu allow you to open a window to execute Macintosh II background tasks, open serial port windows (related to serial port interfaces you create on the MCP-based card), and control the verbosity of coprocessor diagnostic output. The MCP menu includes the commands shown in Figure 17-5.

MCP

Mac II Window

Serial A Window

Serial B Window

Serial Setup

Disable Verbose Messages

Figure 17-5
MCP menu for third-level tests

Mac II Window

The Macintosh II Window command is a background task that operates under MultiFinder; if MultiFinder is not enabled, this command is disabled (dimmed). This command allows you to run a background diagnostic task on the Macintosh II while running third-level diagnostics.

❖ *Note:* This command cannot be used to start a MR-DOS task in the Macintosh II task window.

To use this command to run a background task, you need at least two megabytes of RAM and MultiFinder installed in your Macintosh II computer, in which case the Macintosh II task acts just like any other third-level application. However, the linking of the Macintosh II diagnostic background application to the third-level application libraries is slightly different.

The correct way to generate these background tasks are specified in the makefile examples in the Level 3 Examples folder on the distribution disk.

Serial A Window

This command allows you to open a window to correspond to a Macintosh II serial port.

❖ *Note:* This command is currently not implemented.

Serial B Window

This command allows you to open a window to correspond to a Macintosh II serial port.

❖ *Note:* This command is currently not implemented.

Serial Setup

This command allows the user to set the serial port parameters for the two previous commands.

Disable Verbose Messages

If you do not want to display all of the commands and messages on the screen or send them to the log file, you must first choose Disable Verbose Messages. Only output specified by card tasks using `errprintf()` or `LogError()` will be seen if you select Disable Verbose Messages.

Third-level operations

In normal operation, you will use the `MCP_Diagnostic` main window to download a coprocessor application to the MCP card and run it (as described in the next section). The coprocessor application retrieves commands from the interactive window, performs the appropriate action, and reports the result back to that window. All input to the coprocessor application from the interactive window, as well as output from a coprocessor application to the interactive window, is logged to the file associated with the slot containing the card.

You can also create a stack file to stack commands to be sent to a coprocessor application. When you use this mode, commands to the coprocessor application are retrieved by the `MCP_Diagnostic` and transmitted to the coprocessor application upon a request to the third-level window. You can route output from the coprocessor application to a log file to be examined at a later time.

Coprocessor applications can be terminated by the `MCP_Diagnostic` application, or coprocessor applications themselves may request termination. A coprocessor application can communicate with other coprocessor applications that are currently residing on other MCP cards in the Macintosh II, or cause itself to terminate. In addition, a coprocessor application may send commands that cause another coprocessor application on another card in the Macintosh II to start, process, communicate, or terminate.

Writing coprocessor diagnostics

The coprocessor standalone application tasks you write communicate with the associated `MCP_Diagnostic` window to receive parameters that guide its execution. However, you must first link the communication code module provided in the Level 3 Examples folder on the distribution disk.

Example code written in MPW Assembler and MPW C is provided on the MCP distribution disks. Refer to the example files `echo.c` and `makefile` in the folder `Sequential:Level3 Example:MCP`. For more detailed information on makefiles, refer to the "Advanced Topics" chapter in the MPW reference manual.

Third-level commands that are accepted from a coprocessor application include only the `ReadMessage` and `WriteMessage` commands. However, the following commands can be embedded in the `ReadMessage` and `WriteMessage` commands:

- `DumpRegs`
- `Getmem`
- `Freemem`
- `Kill`
- `Readmem`
- `Run`
- `Send`
- `Writemem`

Each of these commands is described in sections that follow.

When you enter the command `Send` into the active window of the `MCP_Diagnostic` application, you will send `ReadMessage` input to a coprocessor application.

You must incorporate the `WriteMessage` command in your diagnostic application running on the coprocessor on the MCP card to send information to the Macintosh II interface, and the `ReadMessage` command to read any data sent to the application from the Macintosh II user or another diagnostic application. The coprocessor application may output `WriteMessage` data only to the log file or to both the `MCP_Diagnostic` window and the log file. You can see an example of these commands in the example files `stackexecute.c` and `echo.c` in the `Level 3 Examples` folder.

Sending `WriteMessage` data to or from the application is handled by the `MCP_Diagnostic` application, acting for the interactive user or diagnostic application. All ASCII text characters and hexadecimal bytes are permissible.

Level 2 through level 7 interrupt vectors on the card must be handled either by MR-DOS or the diagnostic application as appropriate (refer to Part II of this guide).

The messages and responses sent to or received from the coprocessor application must conform to the third-level message format. Normal termination of a third-level diagnostic occurs when the diagnostic requests that it be terminated, or when an `MCP_Diagnostic` application command requests cancellation of a coprocessor diagnostic.

Creating a stack file

When you use the `Run` command for a coprocessor diagnostic test, the diagnostic can receive its commands and parameters from an input file instead of from its active window.

The `MCP_Diagnostic` application starts the application and waits for a `WriteMessage` command that requests input. After this `WriteMessage` is logged, the application retrieves and executes the next command in the stack file.

All commands except `Send` are echoed to the control window and the log file, then executed. If the next command is a `Send`, then the data of `Send` is sent to the coprocessor application (the `Send` command is also echoed and logged, but is not executed).

If the command execution is successful, the `MCP_Diagnostic` application places a one-character message of `$00` into the third-level application's input buffer to be retrieved by a following `ReadMessage`. If the command is unsuccessful, the `MCP_Diagnostic` application places a one-character message of `$FF` into the third-level application's input buffer to be retrieved by a following `ReadMessage`. At End-of-File on the input stack file, the `MCP_Diagnostic` application places a one-character message of value `$10` into the coprocessor application's input buffer to be retrieved by a following `ReadMessage`.

❖ *Note:* There are no boolean operators in the initial implementation to allow the `MCP_Diagnostic` application to skip or delete commands.

When a stack file is exhausted, the `MCP_Diagnostic` application returns to the active window to process commands for the diagnostic on the card in this slot. You can use a `Kill` command as the last command in a stack file.

❖ *Note:* You must terminate third-level commands in a stack file with an asterisk (*).

The file `stackexecute` in the Level 3 Examples folder shows an example for executing a stack of commands.

Operator commands

The third-level communications shell includes operator commands for controlling tasks in the coprocessor diagnostic; the section on Programmer Subroutines provides functions used by a programmer to write or build tasks for a coprocessor diagnostic. Use the following operator commands for starting, controlling, and stopping tasks running on one or more MCP cards:

- Dumpregs
- Freemem
- Getmem
- Kill
- Readmem
- Run
- Send
- Writemem

In second-level diagnostics, the Macintosh II runs all programs; in the third level, the diagnostics are run by the MCP card, with the Macintosh II handling input, output, and control from the user and the cards. You can enter third-level commands either on the command line, or as part of a stack file.

For some commands, you must specify the name of a coprocessor diagnostic, a stack file, or a log file. The name of a coprocessor diagnostic or a stack file or a log file may be any valid ASCII characters except <, >, [,], {, }, \, (,), and blank. Names are limited to 32 characters and may include the pathname.

- ❖ *Note:* Operator commands are not case sensitive; that is, you may enter `get.mem`, `GETMEM`, or `GetMem`, each with the same results. Use the colon only to designate pathnames.

Dumpregs

The `Dumpregs` command displays the registers, Program Counter, and Status Register for the requesting window or card. The format for the `Dumpregs` command is

```
Dumpregs [-C n]
```

Table 17-1 lists the options for this command.

Table 17-1
Options for third-level `Dumpregs` command

Option	Description
<code>-C n</code>	<code>n</code> is the address of the receiver if other than the active window's slot. Slot numbers include 9, A, B, C, D, E, and M

❖ *Note:* True slot numbers are 9, A, B, C, D and E; M is the Macintosh II task.

Freemem

The `Freemem` command frees a memory block that has been allocated by a `Getmem` command. The format for the `Freemem` command is

```
Freemem -S xxxxxx -L xxxxxx [-C n]
```

Table 17-2 lists the options for this command.

Table 17-2
Options for third-level `Freemem` command

Option	Description
<code>-S xxxxxx</code>	Starting address for card memory in 24-bit Hex "xxxxxx". This address is specified in hexadecimal
<code>-L xxxxxx</code>	Length of card memory expressed as 24-bit Hex value "xxxxxx". This address is specified in hexadecimal
<code>-C n</code>	<code>n</code> is the address of the receiver if other than the active window's slot. Slot numbers include 9, A, B, C, D, E, and M

❖ *Note:* Hexadecimal data is shown in the log file and on the screen window as hex, and is hex for the third-level application program data of `WriteMessage` and `ReadMessage`. All hex digits must be provided, and each byte must have two digits.

Getmem

The `Getmem` command assigns a block of memory on the requesting window or card. The format for the `Getmem` command is

```
Getmem -S xxxxxxx -L xxxxxxx [-C n] [-E]
```

Table 17-3 lists the options for this command.

Table 17-3
Options for third-level `Getmem` command

Option	Description
-S xxxxxxx	Starting address for card memory in 24-bit Hex "xxxxxx". This address is specified in text hex
-L xxxxxxx	Length of card memory expressed as 24-bit Hex value "xxxxxx". This address is specified in text hex
-C n	n is the address of the receiver if other than the active window's slot. Slot numbers include 9, A, B, C, D, E, and M
-E	Requests exclusive use of memory

If you specify the `-E` option, the assigned chunk of memory will not be assigned to any other requestor until after a `Freemem` command is issued against the same block.

If you do not specify the `-E` option, request is for shared use of memory. Shared memory may be assigned to more than one requestor, CPU, or card.

Kill

The `Kill` command terminates execution of a specified coprocessor diagnostic running on this window's slot/card. The format for the `Kill` command is

```
Kill <Level 3 diag name> [-C n] [-DR] [-Y]
```

The coprocessor test name must be specified first, but all following options may be entered in any sequence. Table 17-4 lists the options for this command.

Table 17-4
Options for third-level Kill command

Option	Description
-C <i>n</i>	<i>n</i> is the address of the receiver if other than the active window's slot; slot numbers include 9, A, B, C, D, E, and M
-Y	Indicates <i>Yes</i> , that any previous diagnostic running on this slot may be canceled without interactive agreement from the user
-DR	Displays register contents at time of termination
-A	Kills all diagnostics on all cards

Readmem

The `Readmem` command displays a block of memory on the requesting window or card. This command reads a section of memory and displays it on the screen in the format

```

1000  xxxx xxxx xxxx xxxx xxxx xxxx xxxx xxxx
1010  xxxx xxxx xxxx xxxx xxxx xxxx xxxx xxxx

```

The format for the `Readmem` command is

```
Readmem -S xxxxxx -L xxxxxx [-C n]
```

Table 17-5 lists the options for this command.

Table 17-5
Options for third-level `Readmem` command

Option	Description
<code>-S xxxxxx</code>	Starting address for card memory in 24-bit Hex "xxxxxx"; this address is specified in text hex. Each byte must have two digits
<code>-L xxxxxx</code>	Length of card memory expressed as 24-bit Hex value "xxxxxx"; this address is specified in text hex. Each byte must have two digits
<code>-C n</code>	<i>n</i> is the address of the receiver if other than the active window's slot; slot numbers include 9, A, B, C, D, E, and M

Run

The `Run` command runs a standalone coprocessor diagnostic on the active window's card. The format for the `Run` command is

```
Run <Level 3 diag name> [-C n] [-S <name>] [-O <name>] [-Y] [-L xxxxxx]
```

- ❖ *Note:* Any tasks already running on this window's card requires operator acknowledgment to be killed before running the coprocessor diagnostic. The `MCP_Diagnostic` application calls to your attention any standalone program executing on this window's card before being killed.

The coprocessor test name must be specified first, but all following options may be entered in any sequence. Table 17-6 lists the options for this command.

Table 17-6
Options for third-level `Run` command

Option	Description
<code>-C <i>n</i></code>	<i>n</i> is the address of the receiver if other than the active window's slot; slot numbers include 9, A, B, C, D, E, and M.
<code>-S <<i>name</i>></code>	File that contains the input stack of commands for this third-level diagnostic; stack files may not be shared between cards
<code>-O <<i>name</i>></code>	Directs output from this diagnostic to file < <i>name</i> >
<code>-Y</code>	Indicates <i>Yes</i> , that any previous diagnostic running on this slot may be canceled without interactive agreement from the user
<code>-L <i>xxxxxx</i></code>	Loads diagnostic code onto card memory starting at 24-bit hex address <i>xxxxxx</i> . This address is specified in text hex, and is shown in the log file and on the screen window as text hex; however, it is internal hex for the third-level application program data of <code>WriteMessage</code> and <code>ReadMessage</code> . All hex digits must be provided, and each byte must have two digits.

Send

The `Send` command sends the data between the slot or card. The format for the `Send` command is

```
Send [<Level 3 diag name>] [-C n] "Any ASCII data between double quotes"
```

You must specify the coprocessor test name first, but you can enter all following options in any sequence. Table 17-7 lists the options for this command.

Table 17-7
Options for third-level `Send` command

Option	Description
<code>-C <i>n</i></code>	<i>n</i> is the address of the receiver if other than the active window's slot; slot numbers include 9, A, B, C, D, E, and M
<code>Text</code>	any ASCII data enclosed between double quotation marks
<code>Hexadecimal</code>	data sent embedded in backslashes; for example, <code>\xx xx xx xx\</code>
<code>Text</code>	

Text data is specified in text hex, and is shown in the log file and on the screen window as text hex; however, it is internal hex for the coprocessor application program data of `WriteMessage` and `ReadMessage`. All hex digits must be provided, and each byte must have two digits.

Two double quotation marks in a row sends one double quotation mark as part of the message. For example, if you enter the following statement

```
SEND "Hello ""World""."
```

The message actually sent is

```
Hello "World".
```

Writemem

The `Writemem` command writes data to card memory on the requesting window or card. The format for the `Writemem` command is

```
Writemem -S xxxxxx [-C n] \xxxxxxxxxxxxxxxxxxxxxx\
```

Table 17-8 lists the options for this command.

Table 17-8
Options for third-level `Writemem` command

Option	Description
-S xxxxxx	Starting address for card memory in 24-bit Hex "xxxxxx"; this address is specified in text hex
-C n	n is the address of the receiver if other than the active window's slot; slot numbers include 9, A, B, C, D, E, and M
\xxxxxx\	Hex value to be stored in card memory. This value is specified in text hex. Each byte must have two digits.

Buffer management

Initialization allocates two buffers in Macintosh II memory used for input to the coprocessor diagnostic (output from the MCP_Diagnostic application) and output from the coprocessor diagnostic (input to the MCP_Diagnostic application). The library `taskcommands.c.o` provides routines to initialize, read, and write messages from a diagnostic application on the window's card.

Figure 17-6 illustrates the MCP card communications buffer. Both buffers have head and tail pointers.

MSC NNNN
ART: NN x 17 pi
20.5 pi text to FN b/b

Figure 17-6
MCP card buffer

`StartPtr` points to the start of the buffer and is never modified.

`HeadPtr` points to the current head of the buffer and is maintained by the sender.

`TailPtr` points to the current tail of the buffer and is updated by the receiver.

`Length` is the length of the buffer and is never modified.

The address of the buffers allocated to a coprocessor diagnostic is placed into the card memory at absolute locations `F700081E` (buffer to the coprocessor diagnostic from the MCP_Diagnostic application) and `F7000810` (buffer from the coprocessor diagnostic to the MCP_Diagnostic application).

The supplied routines follow the convention that the head pointer is updated after new data is placed into the buffer by the sender. The routines increment the head pointer by the number of bytes added to the buffer.

The sender is responsible for:

- checking that the data to be placed into the buffer will fit before overrunning the tail pointer
- wrapping around to the start when the end of buffer is encountered

The receiver is responsible for

- updating the tail pointer after retrieving the data from the buffer

If the head pointer is equal to the tail pointer, then the buffer is empty. If the head pointer is one less than the tail pointer, then the buffer is full and cannot receive any new data. These functions are provided in the task command library using the `ReadMessage` and `WriteMessage` functions.

The parameters of a message in the Input/Output buffers is as follows:

Byte 1, Byte 2, Byte 3

Refer to the file `L3c.h` for an example. Table 17-9 lists the format of a message in the Input/Output buffers for coprocessor diagnostic test applications.

Table 17-9
Format of message

Byte	Function	Description
1	Length	The length of all the messages, including the length byte, the command byte, and the data
2	Control	<p>x'80' Message contains an embedded command</p> <p>x'40' Application requests new input</p> <p>x'20' Information request</p> <p>x'10' The Message contains a one-byte response (such as successful completion, EOF on stack file, and so forth)</p> <p>x'04' Message is to be logged only; does not appear on screen of the MCP_Diagnostic application</p> <p>x'02' Message is a WriteMessage to the MCP_Diagnostic application from a card. It prints the message to the screen even if "Verbose data logging" is suppressed</p> <p>x'01' Message is a Send to a card (third-level application)</p> <p>x'00' Null command, message ignored</p>
3 to <i>n</i>	Data	Used for data (messages and requests), where <i>n</i> cannot exceed 255
	Response Message	<p>\$00 Successful execution of a command (except a Send command) from the stack file</p> <p>\$10 End of file on the stack file</p> <p>\$FF Unsuccessful execution of a command (except a Send command) from the stack file</p>
	GetInfo Request	<p>\$01 Send date/time</p> <p>\$02 Send current slot</p> <p>\$03 Send slots with MCP cards in them</p>

Programmer subroutines

The following functions are provided for software running on the MCP to communicate with the coprocessor application. These routines are provided for a programmer who wants to build tasks for a coprocessor diagnostic, and include

- `errprintf()`
- `GetCards()`
- `GetSlot()`
- `GetTimeStamp()`
- `HandleSystemTask()`
- `HexToString()`
- `InitMessage()`
- `KillThisTask()`
- `LogError()`
- `ReadByte()`
- `ReadMessage()`
- `ReadWord()`
- `Reply()`
- `SendNextCommand()`
- `StringToHex()`
- `strlen()`, `strcpy()`, and `strcat()`
- `TickCount()`
- `WriteByte()`
- `WriteMessage()`
- `WriteWord()`

These functions may be used for either the stand-alone task running on the card, or in the Macintosh II background diagnostic task. For more information, see the makefile in the Level 3 Examples folder on the distribution disk.

errprintf()

The function is the same as the `printf` function, but the result is output to the slot's window, regardless of the `Verbose Disabled` command setting. See an example in the source file `echo.c` in the Level 3 Examples folder.

GetCards()

Use the `GetCards()` command to return a short word (16-bit) that corresponds to a set of bit flags (bit numbers 9 through 14) showing which slots have MCP cards in them.

The C format for `GetCards()` is

```
short GetCards ( )
```

The Pascal declaration for `GetCards()` is

```
Function GetCards: Integer; C; External;
```

GetSlot()

Use the `GetSlot()` command to return an integer indicating the slot on which the application is running, listed in Table 17-10.

Table 17-10
Slots for application

Slot Number	Description
9 through E	Application card slot number (slot 9 = 9, slot E = 14)
7	Macintosh task
-1	Error

The C format for `GetSlot()` is

```
GetSlot ( )
```

The Pascal declaration for `GetSlot()` is

```
Function GetSlot: LongInt; C; External;
```

GetTimeStamp ()

Use the `GetTimeStamp ()` command to return the date and time from the Macintosh II memory. This message is returned in a message in the C string format `mm/dd/yy at hh.mm.ss`, where `mm` is the month, `dd` is the day, `yy` is the year, `hh` is the hour, `mm` is the minute, and `ss` is the second.

- ❖ *Note:* Using this command continuously results in very slow performance. It is recommended that you use the `TickCount` command if speed is critical to the performance of your application. You must allocate space for the string before calling this routine.

The C format for `GetTimeStamp ()` is

```
GetTimeStamp (message)
char message[];
```

The Pascal declaration for `GetTimeStamp ()` is

```
Procedure GetTimeStamp (Var message:str255);C;External;
```

HandleSystemTask ()

Occasionally call the `HandleSystemTask ()` command to let the application update internal buffers, as well as to let the shell know that your application is still running.

The C format for `HandleSystemTask ()` is

```
HandleSystemTask ( );
```

The Pascal declaration for `HandleSystemTask ()` is

```
Procedure HandleSystemTask;C;External;
```

HexToString()

Use the `HexToString()` command to convert a binary long number to a text hex string of length digits. The result is placed in memory pointed to by `strPtr`. The length starts from the least significant nibble.

The C format for `HexToString()` is

```
HexToString (number, length, strPtr)
long number;
int length;
char *strPtr;
```

The Pascal declaration for `HexToString()` is:

```
Procedure HexToString (number:LongInt; length:LongInt; strPtr:Ptr); C;
    External;
```

InitMessage()

Use the `InitMessage()` routine before any other to initialize the card to the Macintosh II communications buffer.

The C format for `InitMessage()` is

```
InitMessage (message);
```

The Pascal declaration for `InitMessage()` is

```
Procedure InitMessage;C;External;
```

KillThisTask()

Call the `KillThisTask()` routine when the card diagnostic has completed execution.

The C format for `KillThisTask()` is

```
KillThisTask (message)
```

The Pascal declaration for `KillThisTask()` is

```
Procedure KillThisTask;C;External;
```

LogError()

Use the `LogError()` command to send a message to the screen. This message overrides the data logging suppression and should be used to send messages that will always be logged.

The C format for `LogError()` is

```
LogError (message)
char message[];
```

The Pascal declaration for `LogError()` is

```
Procedure LogError (message:str25); C;External;
```

❖ *Note:* Message is expected to be a C-formatted string.

printf()

This function is the same as defined in the MPW C reference manual, except that it supports only formatted characters, strings, and long integers. To output a character or a short integer, first typecase the integer to long. An example of this can be found in the source file `echo.c` in the Level 3 Examples folder.

ReadByte()

Use the `ReadByte()` command to get a 24-bit address byte from Macintosh II memory.

The C format for `ReadByte()` is

```
char ReadByte (address)
char *address;
```

The Pascal declaration for `ReadByte()` is

```
Function ReadByte (address: Ptr): char; C: External;
```

ReadMessage()

Use the `ReadMessage()` command to read the next input message from the input buffer for this diagnostic application, then place that data into the program buffer `Message`. Data read is only the data that was sent between the double quotation marks, plus the two control bytes. It is the responsibility of the coprocessor diagnostic to allocate space for the message.

The C format for `ReadMessage()` is

```
ReadMessage (Message)
char *Message;
```

The Pascal declaration for `ReadMessage()` is

```
procedure ReadMessage(aString:str255): Integer;C;External;
```

ReadWord()

Use the `ReadWord()` command to get a 16-bit value from an even address on the Macintosh II. The C format for `ReadWord()` is

```
abort
ReadWord (address)
short *address;
```

The Pascal declaration for `ReadWord()` is

```
Function ReadWord (address: Ptr): Integer; C: External;
```

Reply()

Use the `Reply()` command to send a message to the slot's window. This message does not override the data logging suppression and should be used to send messages that are helpful but not essential.

The C format for `Reply()` is

```
Reply (Message)
char *Message;
```

The Pascal declaration for `Reply()` is

```
Procedure Reply (aString:str255) : Integer;C;External;
```

To print a quote, precede it with a backslash. For example, to send the message "Hello, world" to the operator's window in C, enter

```
Reply ("\"Hello, world\" ");
```

SendNextCommand()

Use the `SendNextCommand()` to request that Level 3 read the next command from a third-level stack file. If the command is a `Send`, the contents of the `Send` are returned in message. If not, there are three possible returns, listed in Table 17-11.

Table 17-11
Returns for `SendNextCommand()`

Return	Value	Description
Tabletext	0	The message variable contains the text sent by the operator or stack command
END OF FILE	-1	The stack file is finished and closed
DISK_ERROR	-2	A disk error occurred; stack file processing is disabled
COMMAND_EXECUTED	-3	A stack command other than a <code>Send</code> was executed (that is, <code>GetMem</code> , <code>FreeMem</code> , and <code>DumpRegs</code> commands)

The C format for `SendNextCommand()` is

```
SendNextCommand (message)
char message[];
```

The Pascal declaration for `SendNextCommand()` is

```
Function SendNextCommand (message:str255):LongInt;C;External;
```

StringToHex()

Use the `StringToHex()` command to convert a text hex string into a binary long.

The C format for `StringToHex()` is

```
long StringToHex (strPtr)
char *strPtr;
```

The Pascal declaration for `StringToHex()` is

```
Function StringToHex (strPtr:Str255): LongInt; C; External;
```

strlen(), strcpy(), and strcat()

These commands duplicate the standard C string functions, as defined in the MPW C reference manual.

TickCount()

The C format for `TickCount()` is

```
long TickCount();
```

The Pascal declaration for `TickCount()` is

```
Function TickCount:longInt; C;External;
```

You should not use this command continuously as part of a wait loop, because this will lock up the Macintosh II NuBus.

WriteByte()

Use the `WriteByte()` command to write a byte to the 24-bit address of the Macintosh II memory.

The C format for `WriteByte()` is

```
WriteByte (aByte, address)
char aByte;
char *address;
```

The Pascal declaration for `WriteByte()` is

```
Procedure WriteByte (aByte:char; address: Ptr) C; External;
```

WriteMessage()

Use the `WriteMessage()` command to write a message from the coprocessor diagnostic to the Macintosh interface window.

The C format for `WriteMessage()` is

```
WriteMessage (Message);
```

Any ASCII message will be shown in the window corresponding to this card. You must allocate space for this string before using the function.

The Pascal declaration for `WriteMessage()` is

```
procedure WriteMessage(aString:str255):Integer;C;External;
```

❖ *Note:* This text expects a <format string.

Text hex (ASCII representation of hex) data is sent embedded in backslashes; for example, `\xx xx xx xx\`. This data is shown in the log file and upon the screen window as text hex, but is internal hex for the third-level application program data of `WriteMessage` and `ReadMessage`. If a `WriteMessage` contains an embedded `Send` that contains hex data, that hex data must be in text hex format. Each byte must have two digits.

`WriteMessages` may contain the following as embedded commands:

- Run
- Start
- Send
- Kill
- Memory Commands
- DumpRegs

`WriteMessages` may also contain commands to other slot's diagnostics using the `MCP_Diagnostic` application. Set the embedded command bit as shown in the following example:

Buffer data from Slot 9 :

```
WriteMessage ("\101\202"Run <Levl3Token> -C C -S <TokenCStack> -O <SlotCOutput> ")
```

The `\101` indicates that the total buffer data is of length 65 decimal (101 octal = 65 decimal; 202 octal = \$82 decimal.) The 8 or \$82 specifies a bit telling that the message contains an embedded command. The 2 of \$82 means the message will be displayed to the operator window, even if `VerboseMessages` are suppressed.

The above command embedded in a `WriteMessage` to the `MCP_Diagnostic` application from Slot 1 causes the application to

- run the coprocessor application named `Levl3Token` on the card in slot C;
- feed the application with commands from the file `Token3Stack`; and
- direct all `WriteMessages` to the Slot C screen, as well as to the file `SlotCOutput`.

All commands sent to the application would be echoed to the `SlotCOutput` file.

The `MCP_Diagnostic` application accepts commands for other cards from the interactive screen. The embedded `Run` command is displayed upon the Slot 9 screen as `WriteMessage` data and upon the Slot C screen as a `Run` command for that slot.

WriteWord()

Use the `WriteWord()` command to put a word at an even address of the Macintosh II memory.

The C format for `WriteWord()` is

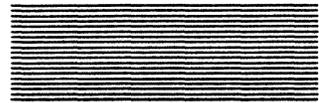
```
WriteWord (aWord,address)
```

```
short aWord;
```

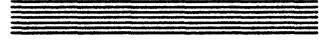
```
short *address;
```

The Pascal declaration for `WriteWord()` is

```
Procedure WriteWord (aWord:Integer;address:Ptr); C; External
```

Appendix A



Files on the MCP Distribution Disks

What this appendix tells you

For your information, this appendix provides a list of folders and files on the MCP distribution disks. Any instructions or references in this guide to files and folders usually include the use for that information in the appropriate chapter; this appendix is for reference and information only. Be sure to check the actual distribution disks for accurate, up-to-the-minute listings of files and folders.

There are currently three distribution disks provided for Version 1.0 of the Macintosh Coprocessor Platform:

- MR-DOS™ 1
- MR-DOS™ 2
- MCP Diagnostics

The contents of each disk are described in the following sections.

Files on MR-DOS™ 1

Table A-1 lists the folders and files found on the distribution disk named MR-DOS™ 1 and provides a brief description of each.

❖ *Note:* The folder name provides a complete description of the pathname to the file.

Table A-1
Files on MR-DOS 1

File name	Description
FOLDER: MR-DOS 1:MR-DOS:	
: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
FOLDER: MR-DOS 1:MR-DOS:AST_ICP	
Download-lib.o	Library containing Download and Findcard subroutines tailored to code to the AST-ICP card
OS.o	Library containing MR-DOS operating system and utility routines tailored to run on the AST-ICP 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 MR-DOS operating system that runs on the AST-ICP card

`osglue.o` Library containing glue code and the `iopruntime` routines for programs that run on the AST-ICP card

FOLDER: MR-DOS 1:MR-DOS:Examples:

`:AST_ICP:` a folder containing example files for the AST-ICP card

`echo.c` The source of an example program of a very simple server task designed to echo messages

`L3MMSVP.a` The source of assembler routines comprising part of the hardware diagnostic task MMSVP
Warning: Diagnostic code is still under development.

`L3MMSVP.c` The source of C routines comprising part of the hardware diagnostic task MMSVP
Warning: Diagnostic code is still under development.

`L3MMSVPClient.c` The source of the task, `L3MMSVPClient`, designed to control the hardware diagnostic task MMSVP
Warning: Diagnostic code is still under development.

`Makefile` MakeFile makes all examples and test tasks found within the `:MR-DOS: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 MR-DOS, initializes any hardware that needs initialization before any tasks start executing, specifies tasks to be initially started when MR-DOS starts executing, and starts MR-DOS executing

`osscoint.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 `osscint.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 an 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

FOLDER: MR-DOS 1:MR-DOS:Examples:AST_ICP:

- `Download` An MPW tool designed to download a module to the AST-ICP card. This module contains the MR-DOS operating system and any tasks or managers that are to be downloaded with the MR-DOS operating system.
- `dumpcard` An MPW tool designed to dump information about an AST-ICP card that is or was running MR-DOS. This tool is meant to assist in trouble-shooting problems. It dumps the global common area of MR-DOS, task control block information for the tasks running under MR-DOS, and other information.
- `echo.c.o` This is the object file of the `echo.c` routine compiled to run on the AST-ICP card
- `L3MMSVP.a.o` The object file of assembler routines comprising part of the hardware diagnostic task MMSVP
Warning: Diagnostic code is still under development.
- `L3MMSVP.c.o` The object file of C routines comprising part of the hardware diagnostic task MMSVP
Warning: Diagnostic code is still under development.
- `L3MMSVPCClient.c.o` The object file of the task, `L3MMSVPCClient`, designed to control the hardware diagnostic task MMSVP
Warning: Diagnostic code is still under development.
- `map` This is the map produced by Link during the building of the start module for the AST-ICP card

`name_tester.c.o` This is the object file of the `name_tester.c` routine compiled to run on the AST-ICP card

`osmain.c.o` This is the object file of the `osmain.c` routine compiled to run on the AST-ICP card

`ossccint.a.o` This is the object file of the `ossccint.a` routine compiled to run on the AST-ICP card

`printf.c.o` This is the object file of the `printf.c` routine compiled to run on the AST-ICP card.

`pr_manager.c.o` This is the object file of the `pr_manager.c` routine compiled to run on the AST-ICP card

`start` This module is produced by Link during the use of the MakeFile in folder :MR-DOS:Examples: for building an example to be downloaded to the AST-ICP card. This module contains the initialization routine `osmain.c`, the version of MR-DOS designed to run on the AST-ICP card, and numerous test tasks.

`timeIt.c.o` This is the object file of the `timeIt.c` routine compiled to run on the AST-ICP card

`timer_tester.c.o` This is the object file of the `timer_tester.c` routine compiled to run on the AST-ICP card

`trace_manager.c.o` This is the object file of the `trace_manager.c` routine compiled to run on the AST-ICP card

`xref` This is the cross reference produced by Link during the building of the `start` module for the AST-ICP card

FOLDER: MR-DOS 1:MR-DOS:Examples:MCP:

`Download` An MPW tool designed to download a module to the MCP card. This module contains the MR-DOS operating system and any tasks or managers that are to be downloaded with the MR-DOS operating system.

`dumpcard` An MPW tool designed to dump information about an MCP card that is or was running MR-DOS. This tool is meant to assist in trouble-shooting problems. It dumps the global common area of MR-DOS, task control block information for the tasks running under MR-DOS, and other information.

`echo.c.o` The object file of the `echo.c` routine compiled to run on the MCP card

<code>L3MMSVP.a.o</code>	The object file of assembler routines comprising part of the hardware diagnostic task MMSVP Warning: Diagnostic code is still under development.
<code>L3MMSVP.c.o</code>	The object file of C routines comprising part of the hardware diagnostic task MMSVP Warning: Diagnostic code is still under development.
<code>L3MMSVPCClient.c.o</code>	The object file of the task, <code>L3MMSVPCClient</code> , designed to control the hardware diagnostic task MMSVP Warning: Diagnostic code is still under development.
<code>map</code>	The map produced by Link during the building of the start module for the MCP card
<code>name_tester.c.o</code>	The object file of the <code>name_tester.c</code> routine compiled to run on the MCP card
<code>osmain.c.o</code>	The object file of the <code>osmain.c</code> routine compiled to run on the MCP card
<code>osscint.a.o</code>	The object file of the <code>osscint.a</code> routine compiled to run on the MCP card
<code>printf.c.o</code>	The object file of the <code>printf.c</code> routine compiled to run on the MCP card
<code>pr_manager.c.o</code>	The object file of the <code>pr_manager.c</code> routine compiled to run on the MCP card
<code>start</code>	The module produced by Link during the use of the MakeFile in folder <code>:MR-DOS:Examples:</code> for building an example to be downloaded to the MCP card; this module contains the initialization routine <code>osmain.c</code> , the version of MR-DOS designed to run on the MCP card, and numerous test tasks
<code>timeIt.c.o</code>	The object file of the <code>timeIt.c</code> routine compiled to run on the MCP card
<code>timer_tester.c.o</code>	The object file of the <code>timer_tester.c</code> routine compiled to run on the MCP card
<code>trace_manager.c.o</code>	The object file of the <code>trace_manager.c</code> routine compiled to run on the MCP card
<code>xref</code>	The cross reference produced by Link during the building of the start module for the MCP card

FOLDER: MR-DOS 1:MR-DOS:includes

<code>clister.h</code>	Include file that defines dummy macros for a program cluster
<code>diags.a</code>	Include file that contains constants used by the hardware diagnostic programs MMSVP and MMSVPClient
<code>diags.h</code>	Include file that contains constants used by the hardware diagnostic programs MMSVP and MMSVPClient
<code>Download.h</code>	Include file that contains constants and definitions used when calling the Download and Findcard subroutines
<code>iccmDefs.a</code>	Include file provided for debugging purposes only that contains constants and definitions used by ICCM
<code>iccmDefs.h</code>	Include file provided for debugging purposes only that contains constants and definitions used by ICCM
<code>managers.a</code>	Include file that contains constants and definitions used when sending message requests to the MR-DOS managers (such as ICCM, Name Manager, and others)
<code>managers.h</code>	Include file that contains constants and definitions used when sending message requests to the MR-DOS managers (such as ICCM, Name Manager, and others)
<code>mrdos.a</code>	Include file that contains constants and definitions used by the MR-DOS operating system, as well as the definition of the global common area
<code>mrdos.h</code>	Include file that contains constants and definitions used by the MR-DOS operating system, as well as the definition of the global common area
<code>os.a</code>	Include file that contains constants and definitions and macros used when invoking MR-DOS primitives (those functions within MR-DOS invoked by instruction traps and include <code>GetMsg</code> , <code>GetMem</code> , <code>Send</code> , <code>Reschedule</code> , and others)
<code>os.h</code>	Include file that contains constants and definitions and external routine declarations used when calling MR-DOS primitives and utility routines (primitives are those functions within MR-DOS invoked by instruction traps and include <code>GetMsg</code> , <code>GetMem</code> , <code>Send</code> , <code>Reschedule</code> , and others; utility routines include <code>GetTID</code> , <code>GetCard</code> , <code>Lookup_Task</code> , and others)
<code>scc.a</code>	Include file that contains the definition of the interrupt handler table used by the routines in :MR-DOS:Examples: that makes use of SCCs

<code>scc.h</code>	Include file that contains the definition of the interrupt handler table used by the routines in <code>:MR-DOS:Examples:</code> that makes use of SCCs
<code>siop.a</code>	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
<code>siop.h</code>	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
<code>timerlibrary.a</code>	Include file that contains the constants and definitions needed to use the timer library
<code>timerlibrary.h</code>	This include file contains the constants and definitions needed to use the timer library

FOLDER: MR-DOS 1:MR-DOS:MCP:

<code>Download-lib.o</code>	Library containing Download and Findcard subroutines tailored to code to the MCP card
<code>OS.o</code>	Library containing MR-DOS operating system and utility routines tailored to run on the MCP card.
<code>OSDefs.d</code>	Assembler symbol table file of <code>mrDOS.a</code> , <code>os.a</code> , <code>managers.a</code> , and <code>siop.a</code> containing constants and macros tailored to the version of the MR-DOS operating system that runs on the MCP card
<code>osglue.o</code>	Library containing glue code and the <code>iopruntime</code> routines for programs that run on the MCP card

Files on MR-DOS 2

Table A-2 lists the folders and files found on the distribution disk named MR-DOS 2 and provides a brief description of each.

Table A-2
Files on MR-DOS 2

File name	Description
-----------	-------------

FOLDER: MR-DOS 2:Apple IPC

<code>: 'Apple IPC' :</code>	Folder containing folders and files for Apple IPC
<code>:Forwarder:</code>	Folder containing the Forwarder application and files

FOLDER: MR-DOS 2:Apple IPC

'Apple IPC'	Contains a driver and code to provide some of the MR-DOS 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

FOLDER: MR-DOS 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

<code>Makefile</code>	Used by <code>Make</code> to create all of the programs and tasks within the :Examples: folder
<code>:MCP:</code>	Folder of examples for the MCP card
<code>name_tester</code>	An MPW tool designed to test the Name Manager
<code>name_tester.c</code>	The source of the test task designed to test the Name Manager
<code>pr_manager</code>	This MPW tool is a Print Manager task. The <code>printf</code> subroutine will look for a Print Manager, and request that a Print Manager print formatted text
<code>pr_manager.c</code>	This is the source of the <code>pr_manager</code> MPW tool.
<code>RSM_File.c</code>	This is the source of a test task <code>RSM_File</code> which is to be dynamically downloaded to a card running MR-DOS.
<code>RSM_tester.c</code>	The source of a MPW tool that dynamically downloads a task to a smart card running MR-DOS
<code>TestR</code>	This MPW tool tests the Apple IPC driver
<code>TestR.c</code>	The source of a MPW tool that tests the Apple IPC driver
<code>timeit</code>	This MPW tool measures the time required to exchange messages between itself and a Echo Manager
<code>timeIt.c</code>	The source of the MPW tool which measures the time required to exchange messages between itself and a Echo Manager
<code>TraceMonitor</code>	This MPW tool receives messages from Trace Managers and records them in a trace file
<code>trace_monitor.c</code>	The source of the <code>TraceMonitor</code> MPW tool
FOLDER: MR-DOS 2:Apple IPC:Examples:AST_ICP	
<code>RSM_File</code>	An example of a module built to be dynamically downloaded to the AST-ICP card
<code>RSM_tester</code>	This MPW tool dynamically downloads a module to the AST- ICP card
FOLDER: MR-DOS 2:Apple IPC:Examples:DumpTrace	
<code>DumpTrace</code>	This MPW tool analyzes a trace file created by the <code>TraceMonitor</code> and dump selected messages from the trace file
<code>dump_16_bytes.c</code>	The source of one of the subroutines comprising the <code>DumpTrace</code> MPW tool
<code>dump_line.c</code>	The source of one of the subroutines comprising the <code>DumpTrace</code> 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: MR-DOS 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: MR-DOS 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)

Files on MCP Diagnostics

Table A-3 lists the folders and files found on the distribution disk named MCP Diagnostics and provides a brief description of each.

Table A-3
Files on MCP Diagnostics

File Name	Description
FOLDER: MCP Diagnostics:MCP_Diagnostics	
Build_Diag_Binary	MPW Script to build all applications in all folders
DiagLib.o	Diagnostic Library used in MCPDiagTool and MCP_Diagnostic
:DiagTool:	Contains MPW diagnostic tool and source code
MCP_Diagnostic	The diagnostic application provided for MCP-based cards
:ROM:	Contains ROM build files and source code
:Sequencer:	Contains user-level diagnostic application and stub files for adding tests
FOLDER: MCP Diagnostics:MCP_Diagnostics:DiagTool:	
'DiagLib Notes'	Describes implementation of DiagLib library
makefile	Make file to generate MCP DiagTool
MCPDiagTool.c	Source to MPW DiagTool
FOLDER: MCP Diagnostics:MCP_Diagnostics:ROM:	
:MCP:	Folder containing the ROM source code ❖ <i>Note:</i> Duplicate this folder and rename for your project before making changes to the contents of this folder.
gang	MPW tool to download ROM application to PROM burner (see the file 'ROM Burn Instructions' for use of this tool)
GetSInfo	Rudimentary application to look for slot manager info
makefile	Makefile to build the MCP ROM application for PROM downloading
PowerOn.a	Generic MCP power-on testing source code; use ApplPoweron.a to add vendor specific power-on tests

PrimaryInit.a Generic MCP Primary Init testing source code; use the file **ApplPrimaryInit.a** to add vendor specific primary initialization tests

FOLDER: MCP Diagnostics:MCP_Diagnostics:ROM:MCP:

application.h Vendor-specific globals and constants

ApplPowerOn.a Used to add on-board vendor power on tests

ApplPrimaryInit.a
Used to add vendor on-board primary initialization tests

ApplROM.a Main line source for ROM code

'ROM Burn Instructions'
Instructions for taking the ROM application and downloading to a Data I/O PROM Burner

FOLDER: MCP Diagnostics:MCP_Diagnostics:Sequencer :

:includes: Directory for includes files

':Level 3 Examples:' Example test code to download and execute programs on the card's 68000 or Mac background task

:MCP: MCP_Diagnostic application folder
❖ *Note:* Copy folder and rename for your own version before making changes.

'Sequencer Errata'
Bugs found in the current release

SequencerLib.o MCP_Diagnostic object library

FOLDER: MCP Diagnostics:MCP_Diagnostics:Sequencer:includes:

Commdeclr.h Miscellaneous constants needed for assembler code

DeclROMEqu.a Declares ROM equates for ROM diagnostics

L3c.h Used for using Pascal routines using C

SlotMgrEqu.a Declares slot equates for ROM diagnostics

FOLDER: MCP Diagnostics:MCP_Diagnostics:Sequencer:'Level 3 Examples':

:MAC_ICP: Contains example task to run as a level 3 background task

:MCP: Contains task to run as a level 3 card task

`echo.c` Example that tests the functionality of the task commands library

`makefile` Creates applications for card and background examples

`stackexecute.c` Example for executing a set of level 3 commands from a file

**FOLDER: MCP Diagnostics:MCP_Diagnostics:Sequencer:'Level 3 Examples':
MAC_ICP:**

Warning

The background task has been tested using version 6.0.2 of the System file; any other system file may cause the system to crash.

`echo` The echo background application

`echo.c.o` Object file for the echo background application

`echo.r.o` Compiled resources for the background task application

`map` Map of link

`printf.c.o` Print library (same as MR-DOS version)

`stackexecute` The background stackexecute application

`stackexecute.c.o` Object file for the background stackexecute application

`stackexecute.r.o` Compiled resource file for stackexecute

`task.r` Resources required for building a background task

`TaskCommands.c.o` Messaging library

**FOLDER: MCP Diagnostics:MCP_Diagnostics:Sequencer:'Level 3 Examples':
MCP:**

`echo` The echo card task application

`hwexceptions.a.o` Used for informing level 3 shell of hardware exceptions

`IOPRuntime.a.o` Initialization library for board task

`map` Map of link

`printf.c.o` Print library (same as MR-DOS version)

`stackexecute` The card task stackexecute application

`TaskCommands.c.o` Initialization and messaging library

FOLDER: MCP Diagnostics:MCP_Diagnostics:Sequencer :MCP:

`Generate_MCP` An MPW script used to build the sample MCP diagnostic

`Samprez.r.o` The MCP diagnostic resource library

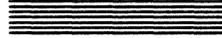
Warning

Although the `Samprez.r.o` file appears as an application, do not run it or your system will crash.

<code>Vendor.a</code>	A stub file for adding in vendor-specific assembler tests
<code>Vendor.r</code>	A stub file for adding in vendor-specific resources
<code>VendorBlocks.p</code>	A stub file for adding in vendor-specific Pascal tests
<code>Vendordefs.p</code>	A stub file for adding in vendor-specific global variables and constants



Appendix B



**Where to go for more
information**

What this appendix tells you

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
<i>Inside Macintosh</i> , 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)
<i>Macintosh Programmer's Workshop (MPW) Reference</i>	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.)
<i>MPW C Language Manual</i>	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)
<i>MPW Assembly Language Manual</i>	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)
<i>Designing Cards and Drivers for Macintosh II and Macintosh SE</i>	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)

List of reference material (continued)

Name	Description
<i>Human Interface Guidelines: The Apple Desktop Interface</i>	Detailed guidelines for developers implementing the Macintosh user interface
<i>Technical Introduction</i>	Introduction to the Macintosh software and hardware to the Macintosh Family 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 the Apple Programmer's and Developer's Association (APDA™).

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

Apple Programmers and Developers Association
 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
<i>Motorola M68000 8-/16-/32-Bit Microprocessors Programmer's Reference Manual</i>	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.

