PRISM SYSTEMS

Mica Working Design Document

Chapter Overviews

digital™

Mica Working Design Document

Chapter Overviews

Digital Equipment Corporation
Confidential and Proprietary

digital™

This information shall not be disclosed to non-Digital Equipment Corporation personnel or generally distributed within Ddigital Equipment Corporation. Distribution is restricted to persons authorized and designated by the responsible Engineer or Manager.

This document shall not be left unattended, and, when not in use, shall be stored in a locked storage area.

These restrictions are to be enforced until noted otherwise.

10 March 1988

Responsible Engineer/Manager *Date*

Document Copy: 71 Date: 3/25/88

Restricted Distribution

Revision History

Date	Revision Number	Author	Summary of Changes
March 1988	1.0	DECwest Engineering	Initial Distribution

TABLE OF CONTENTS

GENERAL

CHAPTER 1 INTRODUCTION TO MICA 1–1

1.1 Overview 1–1
1.1.1 Expandable Base-System Functionality 1–1
1.1.1.1 Object Architecture 1–1
1.1.1.2 Layered I/O System 1–2
1.1.1.3 Protected Subsystems 1–2
1.1.1.4 Client/Server Tools 1–3
1.1.2 Process Structure and Threads 1–3
1.1.3 Cheyenne Database Server 1–3
1.1.4 Glacier Compute Server 1–5

1.2 WDD Structure 1–5

CHAPTER 2 NAMING STANDARDS AND PILLAR CODING STYLE GUIDELINES 2–1

2.1 Overview 2–1
2.1.1 Goals 2–1
2.1.2 Naming Standards 2–1
2.1.3 Pillar Coding Style Guidelines 2–2

CHAPTER 3 STATUS VALUES, MESSAGES, AND TEXT FORMATTING 3–1

3.1 Overview 3–1
3.1.1 Goals 3–1
3.1.2 Status on Mica 3–2
3.1.3 Status Values 3–3
3.1.3.1 SEVERITY Field (bits <2:0>) 3–3
3.1.3.2 MESSAGE_NUMBER Field (bits <15:3>) 3–3
3.1.3.3 FACILITY_NUMBER Field (bits <27:16>) 3–4
3.1.3.4 LOCAL_MESSAGE_NUMBER Field (bits <27:3>) 3–4
3.1.3.5 LOCAL_STATUS Field (bit 28) 3–4
3.1.3.6 FACILITY_SPECIFIC Field (bit 29) 3–4
3.1.3.7 CUSTOMER_FACILITY Field (bit 30) 3–4
3.1.3.8 INHIBIT_MESSAGE_PRINTING Field (bit 31) 3–4
3.1.4 Status and Text Messages 3–4
3.1.4.1 Status Message Format 3–5
3.1.4.2 Message Source Files and Compilation 3–5
3.1.4.3 In-Memory Message Organization 3–5
3.1.4.4 Accessing and Displaying Messages 3–6
3.1.5 Text Formatting 3–6

3.1.6 Open Issues 3–6

EXECUTIVE

CHAPTER 4 THE KERNEL 4–1

4.1 Overview 4–1
4.1.1 Requirements 4–1
4.1.2 Functional Description 4–1
4.1.2.1 Environment of the Kernel 4–1
4.1.2.2 Interaction With the Executive 4–2
4.1.2.3 Primary Kernel Data Structures 4–2
4.1.2.4 Primary Kernel Functions 4–3
4.1.2.5 Performance Data Collection 4–4

CHAPTER 5 OBJECT ARCHITECTURE 5–1

5.1 Overview 5–1
5.1.1 Introduction 5–1
5.1.2 What is an Object? 5–1
5.1.3 Scope 5–1
5.1.4 Requirements and Goals 5–1
5.1.5 Functional Description 5–2
5.1.6 Object-Related Operations 5–4

CHAPTER 6 PROCESS STRUCTURE 6–1

6.1 Overview 6–1
6.1.1 Goals/Requirements 6–1
6.1.2 UJPT Hierarchy 6–1
6.1.2.1 The User Object 6–1
6.1.2.1.1 Functional Interface 6–1
6.1.2.2 The Job Object 6–1
6.1.2.2.1 Functional Interface 6–2
6.1.2.3 The Process Object 6–2
6.1.2.3.1 Functional Interface 6–2
6.1.2.4 The Thread Object 6–2
6.1.2.4.1 Functional Interface 6–2
6.1.3 UJPT Setup/Teardown 6–3
6.1.3.1 UJPT Setup 6–3
6.1.3.2 UJPT Teardown 6–3

CHAPTER 7 MEMORY MANAGEMENT 7–1
7.1 Overview 7–1
7.1.1 Requirements 7–1
7.1.2 Functional Description 7–1
7.1.2.1 Environment of Memory Management 7–1
7.1.3 Memory Management Data Structures 7–2
7.1.4 Differences from the VAX/VMS Memory Management Subsystem 7–4

CHAPTER 8 I/O ARCHITECTURE 8–1
8.1 Overview 8–1
8.1.1 Function Processors 8–1
8.1.2 Objects Used by the I/O System 8–1
8.1.2.1 FPU Object 8–2
8.1.2.2 Channel Object 8–2
8.1.2.3 FPD Object 8–2
8.1.3 I/O Request Synchronization 8–2
8.1.4 I/O Service Routines 8–2
8.1.5 I/O Security 8–3

CHAPTER 9 SYSTEM SERVICE ARCHITECTURE 9–1
9.1 Overview 9–1
9.1.1 Functional Description 9–1
9.1.1.1 System Service Dispatcher 9–2
9.1.1.2 The System Service 9–3
9.1.1.3 System Service Completion 9–3
9.1.1.4 Repeatable and Resumable System Services 9–3
9.1.2 Changes to the Existing Chapter 9–4

CHAPTER 10 SECURITY AND PRIVILEGES 10–1
10.1 Overview 10–1
10.1.1 Authentication 10–1
10.1.2 Access Control 10–2
10.1.2.1 User Access Rights 10–2
10.1.2.2 Object Access Control Information 10–2
10.1.2.3 Access Control Algorithm 10–3
10.1.3 Security Audits 10–3
10.1.4 Issues 10–3

CHAPTER 11 CONDITION, EXIT, AND AST HANDLING 11–1
11.1 Overview 11–1
11.1.1 Condition Handling 11–1
11.1.2 Unwinding 11–2
11.1.3 Exit Handling 11–3
11.1.4 User-Mode AST Handling 11–3
11.1.5 Dependencies 11–4

CHAPTER 12 BOOTING 12–1
12.1 Overview 12–1
12.1.1 Bootstrap Structure 12–1
12.1.2 Hardware Bootstrap 12–1
12.1.3 Primary Software Bootstrap 12–2
12.1.4 Secondary Software Bootstrap 12–2
12.1.4.1 Ultrix Secondary Bootstrap 12–2
12.1.4.2 Mica Secondary Bootstrap 12–3
12.1.5 Mica Bootstrap Summary 12–3

CHAPTER 13 SYSTEM DUMP ANALYZER AND SYSTEM DEBUGGER 13–1
13.1 Overview 13–1
13.1.1 System Dump Analyzer 13–1
13.1.1.1 Requirements & Goals 13–1
13.1.1.2 Design Highlights 13–2
13.1.1.3 Issues 13–2
13.1.2 System Debugger 13–3
13.1.2.1 Requirements & Goals 13–3
13.1.2.2 Design Highlights 13–3
13.1.2.3 Issues 13–4

EXECUTIVE ROUTINES

CHAPTER 14 EXECUTIVE ROUTINES 14–1
14.1 Overview 14–1
14.1.1 System Routines 14–1
14.1.2 Executive Routine Interface Guidelines 14–2
14.1.2.1 General Guidelines 14–2
14.1.2.1.1 Parameter Options 14–2
14.1.2.1.2 Parameter Ordering 14–2
14.1.2.1.3 Parameter Types 14–3
14.1.2.1.3.1 Record Types 14–3
14.1.2.1.4 Return Value 14–3
14.1.2.2 Object Service Routines 14–4
14.1.2.2.1 Object Creation Executive Routines 14–4
14.1.2.2.2 Get/Set Information Executive Routines 14–5
14.1.2.3 General Executive Routines 14–5

14.1.3 System Service Definitions 14–5

I/O

CHAPTER 15 DIRECT ACCESS MASS STORAGE FUNCTION PROCESSORS 15–1

15.1 Overview 15–1
15.1.1 Goals 15–1
15.1.2 Disk Logical Block Interface 15–2
15.1.3 Function Processors 15–3
15.1.3.1 MSCP Function Processor 15–3
15.1.3.2 Disk Striping Function Processor 15–3
15.1.3.3 Disk Shadowing Function Processor 15–4
15.1.4 Error Handling and Diagnostics 15–5
15.1.4.1 Invalid I/O Request 15–5
15.1.4.2 Power Failure 15–5
15.1.5 Sample I/O Request Flow 15–5

CHAPTER 16 MAGNETIC TAPE FUNCTION PROCESSORS 16–1

16.1 Overview 16–1
16.1.1 Goals and Requirements 16–1
16.1.2 Tape Logical-Block Interface 16–2
16.1.3 TMSCP Class Function Processor 16–3

CHAPTER 17 SYSTEM COMMUNICATION SERVICES 17–1

17.1 Overview 17–1
17.1.1 Goals and Requirements 17–1
17.1.2 SCS Functionality 17–1
17.1.3 Implementation Strategy 17–2
17.1.3.1 Initialization and System/Path Recognition 17–2
17.1.3.2 Message and Datagram Buffer Allocation 17–3
17.1.3.3 SYSAP-SCS Interface 17–3
17.1.3.4 SCS-Device Function Processor Interface 17–4
17.1.3.5 Flow Control Scheme 17–4
17.1.3.5.1 SCS Protocol Messages 17–5
17.1.3.5.2 SCS Application Messages and Block Data Transfers 17–5
17.1.3.6 Error Philosophy 17–5

CHAPTER 18 XCA FUNCTION PROCESSOR 18–1

18.1 Overview 18–1
18.1.1 Introduction 18–1
18.1.2 Requirements 18–1
18.1.3 Goals 18–1
18.1.4 Functionality 18–2
18.1.5 Higher-level Interface to XCA Function Processor 18–2
18.1.6 XCA Function Processor Interface to the XCA Port 18–3
18.1.7 XCA Function Processor Implementation 18–4
18.1.7.1 System Recognition 18–4
18.1.7.2 Virtual Circuits 18–4
18.1.7.3 Response Handling 18–5
18.1.7.4 Error Handling 18–5

CHAPTER 19 NI FUNCTION PROCESSOR 19–1

19.1 Overview 19–1
19.1.1 Goals 19–1
19.1.2 Features Not Implemented 19–1
19.1.3 Capabilities 19–1
19.1.4 Interface with the Upper Layer 19–2
19.1.4.1 Request and Execute I/O Functions 19–2
19.1.4.2 Synchronous I/O Call Functions 19–3
19.1.4.3 Callbacks 19–3
19.1.5 Implementation Strategy 19–3
19.1.5.1 Transmit 19–4
19.1.5.2 Receive 19–5
19.1.6 Outstanding Issues 19–6

CHAPTER 20 CONSOLE SUPPORT 20–1

20.1 Overview 20–1
20.1.1 Requirements 20–2
20.1.2 Design Highlights 20–2
20.1.2.1 Console Terminal 20–2
20.1.2.1.1 Synchronous Interface 20–3
20.1.2.1.2 Asynchronous Interface 20–3
20.1.2.2 Console Storage Device 20–3
20.1.2.3 Configuration Processor 20–4
20.1.3 Issues 20–4

CHAPTER 21 MESSAGE FUNCTION PROCESSOR 21–1

21.1 Overview 21–1
21.1.1 Functionality 21–1
21.1.2 Design 21–3
21.1.3 Functional Interface 21–3

CHAPTER 22 PRISM DIAGNOSTIC MONITOR 22–1

22.1 Overview 22–1
22.1.1 Introduction 22–1
22.1.2 Diagnostic Run-time Environments 22–1
22.1.3 Functional Overview 22–2
22.1.4 Components of a PDM-based Diagnostic Program 22–4
22.1.5 PDM Design Goals 22–4
22.1.6 PDM Design Non-goals 22–4
22.1.7 Requirements on Other Products for Meeting Design Goals 22–5
22.1.8 PDM Interfaces 22–5
22.1.8.1 User Interface 22–5
22.1.8.2 Programmer Interface—Diagnostic Services 22–6
22.1.9 PDM Internal Interfaces 22–6
22.1.9.1 Interface Between the User Interface and the PDM Server 22–6
22.1.9.2 PDM/Diagnostic Interface 22–6
22.1.10 PDM's Interfaces to the On-line and Off-line Environments 22–6
22.1.11 Other PDM Features 22–6
22.1.12 Security Issues 22–7
22.1.13 Changes from Rev. 1.0 of "The PRISM Diagnostic Environment" 22–7

CHAPTER 23 ERROR LOGGING 23–1

23.1 Overview 23–1
23.1.1 What is Error Logging? 23–1
23.1.2 How Is Error Information Stored? 23–1
23.1.3 What Does the System Error Log File Contain? 23–1
23.1.4 What Do System Error Log Records Contain? 23–2
23.1.5 Who Creates System Error Log Records? 23–2
23.1.6 How are Records Placed into the Error Log File? 23–2
23.1.7 How Can the System Error Log File be Read? 23–2
23.1.8 Where Does ERF Reside and Execute? 23–2
23.1.9 Where Do ERF Users Reside? 23–3
23.1.10 Who can Access the System Error Log File? 23–3
23.1.11 Who can Control Error Logging? 23–3
23.1.12 How Is the Error Log Data Used? 23–3
23.1.13 How Does Off-line Error Logging Work? 23–3
23.1.14 Are There Other Error Logging Facilities? 23–3

FILE SYSTEM

CHAPTER 24 DISK FILE SYSTEM FUNCTION PROCESSORS 24–1

24.1 Overview 24–1
24.1.1 Files and Directories 24–1
24.1.2 Volume Sets 24–2
24.1.3 Objects Used By Disk File System Function Processors 24–2
24.1.3.1 Function Processor Unit Objects 24–2
24.1.3.2 Channel Objects 24–3
24.1.4 Other I/O Architecture Support 24–3
24.1.5 Disk File System Class Interface Functions 24–3
24.1.6 Other Topics 24–4

CHAPTER 25 FILES–11 ODS2 FUNCTION PROCESSOR 25–1

25.1 Overview 25–1
25.1.1 Files–11 ODS2-3 Data Structures 25–1
25.1.2 Threads 25–3
25.1.3 FPU procedures 25–3
25.1.4 Mounting a volume 25–4
25.1.5 Dismounting a volume 25–4
25.1.6 Volume Sets 25–5
25.1.7 Object Names 25–5
25.1.8 Access Matrix 25–5
25.1.9 Security 25–5
25.1.10 Mapping & Retrieval Pointers 25–5
25.1.11 Read/Write 25–5
25.1.12 Caches 25–6
25.1.13 Other Topics 25–6

CHAPTER 26 RECORD MANAGEMENT SERVICES 26–1

26.1 Overview 26–1
26.1.1 RMS Functionality 26–1
26.1.2 RMS Programming Interface 26–3
26.1.3 Sample I/O Request Flow 26–4

CHAPTER 27 CACHING 27–1

27.1 Overview 27–1
27.1.1 Issues Related to the Design of a System-wide Data Cache 27–1
27.1.2 Summary 27–2

CHAPTER 28 FILE MANAGEMENT UTILITIES 28–1

28.1 Overview 28–1
28.1.1 Goals 28–1
28.1.2 Requirements for the File Management Utilities 28–1
28.1.3 Utilities in the Off-line System 28–2
28.1.4 Integration with System Management 28–2
28.1.5 Description of the Utilities 28–2
28.1.5.1 The Initialize Utility 28–2
28.1.5.2 The Mount Utility 28–3
28.1.5.3 The Dismount Utility 28–3
28.1.5.4 The Verify Utility 28–3
28.1.5.5 The Backup Utility 28–4

IMAGE RELATED

CHAPTER 29 OBJECT MODULE AND IMAGE FILE FORMAT 29–1

29.1 Overview 29–1
29.1.1 Requirements 29–1
29.1.2 Description 29–2
29.1.3 Dependencies 29–4

CHAPTER 30 LINKER 30–1

30.1 Overview 30–1
30.1.1 Requirements 30–1
30.1.2 Implementation 30–2
30.1.2.1 Initial Stage 30–2
30.1.2.2 Pass 1 30–2
30.1.2.3 Intermediate 30–2
30.1.2.4 Pass 2 30–2
30.1.2.5 Final Stage 30–3
30.1.3 Compiler Dependency 30–3

CHAPTER 31 IMAGE ACTIVATION 31–1

31.1 Overview 31–1
31.1.1 Goals/Requirements 31–1
31.1.2 Functional Description 31–1
31.1.2.1 Image Initialization 31–1
31.1.2.2 Image Exit 31–2
31.1.2.3 Autoload Procedure 31–2
31.1.2.4 Installation of Images 31–2
31.1.2.5 Images Within Shareable Image Space 31–3
31.1.3 Issues to be Resolved 31–3

SYSTEM MANAGEMENT AND ADMINISTRATION

CHAPTER 32 SYSTEM MANAGEMENT 32–1
32.1 Overview 32–1
32.1.1 Functional Description 32–1
32.1.2 System Management Model 32–2
32.1.2.1 The System Management User Interface 32–2
32.1.2.2 The System Management Server 32–3
32.1.3 RPC Interface 32–3
32.1.4 Managing Multiple Systems 32–3
32.1.4.1 Glacier Systems 32–4
32.1.4.2 Cheyenne Systems 32–4
32.1.5 Security 32–4
32.1.6 Subset System Management Access 32–4
32.1.7 Authorization, Proxy, Identifier, and Startup Parameter Files 32–4
32.1.8 Server Design Considerations 32–5
32.1.9 Issues 32–5

CHAPTER 33 OPERATOR COMMUNICATIONS 33–1
33.1 Overview 33–1
33.1.1 Functional Description 33–1
33.1.2 OPCOM Components 33–2
33.1.2.1 Client System Management Interface 33–3
33.1.2.2 Client Operator Display Process 33–3
33.1.2.3 Client Operator Request Program 33–3
33.1.2.4 OPCOM Server 33–4
33.1.2.5 Mica Message Function Processor Units (FPUs) 33–4
33.1.2.6 Reader Threads 33–5
33.1.3 AIA Functionality 33–5
33.1.4 Native mode OPCOM calls 33–5
33.1.5 Manipulating log files on Cheyenne 33–5
33.1.6 Issues 33–6

CHAPTER 34 CONFIGURATION MANAGEMENT SOFTWARE 34–1
34.1 Overview 34–1
34.1.1 Goals 34–1
34.1.2 Functional Description 34–1
34.1.2.1 Actions at System Boot Time 34–2
34.1.2.2 Actions During Normal Operation 34–2
34.1.2.2.1 The Error-Monitor Process 34–2
34.1.2.2.2 The Configuration-Manager Process 34–3
34.1.2.2.3 Configuration Function Processor 34–3
34.1.3 Design 34–4
34.1.3.1 The Error-Monitor Process 34–4
34.1.3.2 The Configuration-Manager Process 34–5
34.1.3.3 The Configuration Function Processor 34–6

34.1.4 Relation to Other Software 34–6
34.1.4.1 Memory Management 34–6
34.1.4.2 Shadow Function Processor 34–6
34.1.4.3 MSCP Function Processor and Controller Function Processor 34–6
34.1.4.4 SCS Function Processor 34–6
34.1.4.5 Device Function Processors 34–6
34.1.4.6 Machine Check 34–7
34.1.4.7 Console Software 34–7
34.1.4.8 External Service Processor 34–7
34.1.5 Issues 34–7

CHAPTER 35 SYSTEM VOLUME LAYOUT AND SOFTWARE INSTALLATION 35–1

35.1 Overview 35–1
35.1.1 System Volume Layout 35–1
35.1.1.1 The Read-Only Area 35–1
35.1.1.2 The Read/Write Area 35–1
35.1.1.3 Read-Only and Read/Write Area Interaction 35–2
35.1.2 Software Installation 35–2
35.1.2.1 Goals 35–2
35.1.2.2 General Description 35–3
35.1.2.3 Standard Installation 35–3
35.1.2.3.1 Initial Installation 35–3
35.1.2.3.2 Update Installation 35–3
35.1.2.4 Special Installation 35–3
35.1.2.4.1 Special Installation Types 35–4
35.1.2.4.2 Special Installation Procedure 35–4
35.1.2.5 Front-End and Client Software Installation 35–5
35.1.2.6 High-Availability Configuration 35–5

TESTING AND PERFORMANCE MEASUREMENT

CHAPTER 36 PERFORMANCE MONITOR 36–1

36.1 Overview 36–1
36.1.1 Goals 36–1
36.1.2 Terminology 36–1
36.1.3 Functional Overview 36–1
36.1.3.1 Time Intervals 36–2
36.1.3.2 Classes 36–2
36.1.4 Implementation Overview 36–2
36.1.5 Issues 36–3

CHAPTER 37 USER-LEVEL SYSTEM EXERCISER 37-1

37.1 Overview 37-1
37.1.1 Goals 37-1
37.1.2 Non-Goals 37-1
37.1.3 Outline of the Functionality 37-2
37.1.3.1 Interactions with Other Software 37-2
37.1.4 Outline of the Design 37-3
37.1.4.1 Input and Initialization 37-3
37.1.4.2 Device Testing 37-3
37.1.4.3 Load and Application Specific Testing 37-3
37.1.4.3.1 Testing Glacier 37-3
37.1.4.3.2 Fault Tolerant Testing 37-3
37.1.5 Developing Glacier User Tests 37-3
37.1.6 Requirements 37-4
37.1.6.1 User Diagnostics Interface 37-4
37.1.6.2 Error Logging and Symptom Directed Diagnostics 37-4
37.1.6.3 Mica System Services 37-4
37.1.7 Open Issues 37-4

NETWORK

CHAPTER 38 MICA NETWORK OVERVIEW 38-1

38.1 Overview 38-1
38.1.1 Requirements 38-3
38.1.2 Goals 38-4
38.1.3 Nongoals 38-4
38.1.4 Network Software Components 38-5
38.1.4.1 Data Link 38-5
38.1.4.2 Transports 38-6
38.1.4.3 Value-Added Services 38-6
38.1.4.4 Applications 38-6

CHAPTER 39 NETWORK SERVICES 39-1

39.1 Overview 39-1
39.1.1 Requirements and Goals 39-1
39.1.2 DNA Components 39-2
39.1.3 User Interface 39-2
39.1.4 Implementation 39-4

CHAPTER 40 DNA NAMING SERVICE CLERK 40-1

40.1 Overview 40-1
40.1.1 Requirements 40-2
40.1.2 Functional Interfaces 40-2
40.1.3 Implementation 40-2

CHAPTER 41 DECNET STARTUP, SHUTDOWN, MANAGEMENT, AND LOGGING 41–1

41.1 Overview 41–1
41.1.1 Requirements 41–1
41.1.2 Network Management and Event Logging 41–1
41.1.2.1 Entities, Directors, and Agents 41–1
41.1.2.2 Node Entity 41–4
41.1.2.3 CMIP and the CMIP Server 41–4
41.1.2.4 DECnet–Mica Event Dispatcher 41–5
41.1.3 Network Management Security 41–5
41.1.4 DECnet–Mica Startup 41–5
41.1.5 DECnet–Mica Shutdown 41–6
41.1.6 Issues 41–6

CHAPTER 42 QUARTZ INTERPROCESS COMMUNICATION 42–1

42.1 Overview 42–1
42.1.1 Requirements/Goals 42–1
42.1.2 Non-Goals 42–1
42.1.3 Functional Description 42–1
42.1.4 Design 42–2
42.1.4.1 Message Region Object 42–2
42.1.4.1.1 Functional Interface 42–2
42.1.4.2 Message Queue Object 42–3
42.1.4.2.1 Remote Queues 42–5
42.1.4.2.2 Functional Interface 42–6
42.1.4.3 Message Gate Objects 42–6
42.1.4.3.1 Functional Interface 42–6

DISTRIBUTED FILE SERVICES

CHAPTER 43 DISTRIBUTED FILE SERVICE INTRODUCTION 43–1

43.1 Overview 43–1
43.1.1 Goals 43–1
43.1.2 Model 43–1
43.1.3 Components 43–2
43.1.4 Planned Restrictions 43–4
43.1.5 Network Transparency 43–4
43.1.5.1 Naming 43–5
43.1.5.2 Security 43–5

CHAPTER 44 DISTRIBUTED FILE SERVICE MANAGEMENT 44–1

44.1 Overview 44–1
44.1.1 Restricting Access to Management Operations 44–1
44.1.2 Startup 44–2
44.1.3 Monitoring 44–2
44.1.4 DECnet Name Service 44–2
44.1.5 Management of Files Accessed Through DFS 44–2
44.1.6 System Management 44–2
44.1.6.1 Client Function Processor 44–3
44.1.6.2 Server 44–3
44.1.6.3 Request/Response Function Processor 44–3

CHAPTER 45 DISTRIBUTED FILE SERVICE COMMUNICATION FUNCTION PROCESSOR 45–1

45.1 Overview 45–1
45.1.1 RR 45–1
45.1.1.1 Interface to Higher-Level Function Processors and Threads 45–2
45.1.1.2 Interface to DECnet Session Layer 45–2
45.1.1.3 Interface to System Management 45–3
45.1.1.4 Implementation 45–3
45.1.1.4.1 Data Structures 45–3
45.1.1.4.2 Design Considerations and Issues 45–3
45.1.2 RCL 45–4
45.1.2.1 Structure of RCL Messages 45–4
45.1.2.2 Subroutines 45–5

CHAPTER 46 DISTRIBUTED FILE SERVICE CLIENT FUNCTION PROCESSOR 46–1

46.1 Overview 46–1
46.1.1 Requirements 46–1
46.1.2 Functional Interface 46–1
46.1.3 Internal Design 46–2

CHAPTER 47 DISTRIBUTED FILE SERVICE SERVER 47–1

47.1 Overview 47–1
47.1.1 Sessions 47–1
47.1.2 Server Process Implementation 47–1
47.1.3 File Protocol 47–2
47.1.4 Security 47–2
47.1.5 Caching 47–2
47.1.6 Buffering 47–2
47.1.7 Accounting and Quota Enforcement 47–3
47.1.8 Failure Recovery 47–3

DATABASE SERVER

CHAPTER 48 CHEYENNE OVERVIEW 48–1

48.1 Overview 48–1
48.1.1 Product Goals 48–2
48.1.1.1 Data Integrity 48–3
48.1.1.2 Reliability and Availability 48–3
48.1.1.3 Performance 48–3
48.1.2 Components 48–4
48.1.2.1 Stone 48–4
48.1.2.2 Extended Service Processor 48–5
48.1.2.3 Mass Storage 48–5
48.1.2.4 Mica 48–5
48.1.2.5 Quartz 48–5
48.1.2.6 Client Software 48–6
48.1.2.6.1 Communications 48–6
48.1.2.6.2 Mica and Quartz System Management and Database Administration . 48–6
48.1.2.6.3 Security 48–6
48.1.2.6.4 Database Tools 48–7

48.2 Target Customer Base 48–7
48.2.1 Application Users 48–8
48.2.2 Application Writers 48–8
48.2.3 Database Administrators 48–8
48.2.4 System Managers 48–9
48.2.5 Operations Staff 48–9
48.2.6 Software Support Personnel 48–9
48.2.7 Hardware Service Personnel 48–10

48.3 Hardware Components 48–10
48.3.1 Client Systems 48–10
48.3.2 Standard Configurations 48–10
48.3.3 Highly Available Configurations 48–11
48.3.4 Mass Storage 48–11

48.4 Software Components 48–11
48.4.1 Components on Client Systems 48–11
48.4.1.1 Access to Cheyenne Databases 48–12
48.4.1.2 System Management and Database Administration 48–13
48.4.1.3 Database Tools 48–14
48.4.1.4 Communications 48–14
48.4.2 Components on Stone Systems 48–15
48.4.2.1 Quartz 48–15
48.4.2.2 Mica Executive 48–16
48.4.2.3 System Management 48–16
48.4.2.4 Network Management 48–17
48.4.2.5 Transaction Management 48–18
48.4.2.6 Cheyenne Diagnosis and Maintenance 48–18

48.5 Special Challenges 48–20

48.5.1 Achieving High Availability 48–20
48.5.2 Support 48–21
48.5.3 Testing 48–22
48.5.4 Ease of Use and Internationalization Requirements 48–22

48.6 Related Products 48–23
48.6.1 Other DIGITAL Products 48–23
48.6.2 Future Versions of Cheyenne 48–24

48.7 Issues and TBD 48–25

CHAPTER 49 TRANSACTION SERVICES 49–1

49.1 Overview 49–1
49.1.1 Goals 49–1
49.1.2 Functional Overview 49–2
49.1.2.1 Transaction Object Service Routines 49–3
49.1.2.2 Recovery Manager 49–4
49.1.3 Algorithms 49–4
49.1.3.1 Redo and Undo/Redo Logging 49–4
49.1.3.2 Two-Phase Commit with Presumed Abort 49–4
49.1.3.3 Other Techniques 49–5
49.1.4 Issues 49–5
49.1.5 Bibliography 49–6

COMPUTE SERVER

CHAPTER 50 GLACIER OVERVIEW 50–1

50.1 Overview 50–1
50.1.1 Goals 50–1
50.1.1.1 Client/Server Integration 50–1
50.1.1.2 Application Integration Architecture 50–2
50.1.1.3 Multiple Operating Systems Support 50–2
50.1.1.4 Client Modification 50–2

50.2 Target Customer Base 50–2
50.2.1 Application Users 50–2
50.2.2 Application Developers 50–3
50.2.3 System Managers 50–3
50.2.4 Operations Staff 50–3
50.2.5 Software Support Personnel 50–3
50.2.6 Hardware Service Personnel 50–4
50.2.7 Internal Software Developers 50–4

50.3 First Revenue Ship Applications 50–4

50.4 Glacier Components 50–4
50.4.1 Client Hardware Components 50–4

50.4.2 Client Software Components 50–5
50.4.2.1 Software Run-Time Environment 50–5
50.4.2.2 Software Development Environment 50–6
50.4.2.3 System Management 50–6
50.4.2.4 Underlying Software Mechanisms 50–7
50.4.2.4.1 Network Support 50–7
50.4.2.4.2 Remote Procedure Calls (RPC) 50–7
50.4.2.4.3 Served Disks 50–7
50.4.3 Server Hardware Components 50–7
50.4.3.1 FRS Hardware Configuration 50–8
50.4.3.2 Follow-On Configuration 50–9
50.4.4 Server Software Components 50–9
50.4.4.1 Software Run-Time Environment 50–9
50.4.4.1.1 Application Integration Architecture 50–9
50.4.4.1.2 Application Migration 50–10
50.4.4.1.3 Record Management Services 50–10
50.4.4.2 Software Development Environment 50–10
50.4.4.2.1 Program Development Tools 50–11
50.4.4.3 System Management 50–12
50.4.4.3.1 System Management Server 50–12
50.4.4.3.2 Performance Monitor 50–12
50.4.4.3.3 Console Support 50–12
50.4.4.3.4 System Dump Analyzer 50–12
50.4.4.3.5 Error Logging 50–12
50.4.4.4 Underlying Software Mechanisms 50–13
50.4.4.4.1 Mica Operating System 50–13
50.4.4.4.2 DECnet-Mica Phase V 50–13
50.4.4.4.3 Remote Procedure Calls 50–13
50.4.4.4.4 Distributed File Services (DFS) 50–14
50.4.4.4.5 Job Controller Server 50–14

50.5 Special Challenges 50–14

50.6 Outstanding Issues 50–14

CHAPTER 51 MICA COMPUTE SERVER SUPPORT 51–1

51.1 Overview 51–1
51.1.1 Goals 51–1
51.1.1.1 Activation of a Mica Image 51–2
51.1.1.2 Application Integration Architecture 51–2
51.1.1.3 Support for Development Tools 51–2
51.1.2 Components 51–2
51.1.2.1 Mica Components 51–3

CHAPTER 52 VMS COMPUTE SERVER SUPPORT 52–1
52.1 Overview 52–1
52.1.1 Requirements 52–1
52.1.2 Assumptions 52–2
52.1.3 Functional Description 52–2
52.1.3.1 Image Activation 52–2
52.1.3.2 RPC Calls for VMS Services 52–2
52.1.3.3 Condition Handling 52–2
52.1.3.4 Termination 52–2
52.1.3.5 Debugger Support 52–2

CHAPTER 53 ULTRIX COMPUTE SERVER SUPPORT 53–1
53.1 Overview 53–1
53.1.1 Goals 53–1
53.1.1.1 Execution of a Mica Image 53–1
53.1.1.2 Access to the Client Environment. 53–1
53.1.1.3 Development Tool Support 53–2
53.1.2 Functional Description 53–2
53.1.2.1 The Client Context Server 53–3
53.1.2.2 Mica Program Development on ULTRIX 53–4

CHAPTER 54 PROTECTED SUBSYSTEMS AND RPC 54–1
54.1 Overview 54–1
54.1.1 Goals 54–1
54.1.1.1 Functionality for Mica System Components 54–2
54.1.1.2 Functional Basis for Protected Subsystems 54–3
54.1.1.3 Easy Migration to Corporate RPC 54–3
54.1.1.4 Hide RPC Usage Behind the Stub Generator 54–3
54.1.2 Nongoals 54–4
54.1.2.1 Customer Visibility 54–4
54.1.2.2 All-Encompassing Mechanism 54–4
54.1.2.3 Interoperation with other RPC protocols 54–4
54.1.3 Communications Transport 54–5
54.1.4 Issues 54–5

CHAPTER 55 RPC STUB GENERATOR 55–1
55.1 Overview 55–1
55.1.1 Requirements, Goals, and Nongoals 55–2
55.1.1.1 Requirements 55–3
55.1.1.2 Goals 55–3
55.1.1.3 Nongoals 55–3
55.1.2 Operation of the Stub Generator 55–3
55.1.3 Implementation Strategy 55–4
55.1.4 Dependencies 55–5
55.1.5 Long-Term Mica RPC Stub Generator Strategy 55–5

CHAPTER 56 AIA USER INTERFACE 56–1

56.1 Overview 56–1
56.1.1 Goals 56–1
56.1.2 DECwindows 56–1
56.1.2.1 The X Window System 56–2
56.1.2.1.1 DECwindows Server and Device Drivers 56–3
56.1.2.1.2 Network Protocol and Transport Mechanism 56–3
56.1.2.1.3 Xlib and Xtoolkit Programming Libraries 56–3
56.1.2.2 Application Programming Libraries 56–3
56.1.2.2.1 The DECtoolkit 56–3
56.1.2.2.2 DDIF Toolkit 56–3
56.1.2.3 Implementation Strategy 56–4
56.1.2.4 Dependencies 56–4

CHAPTER 57 MISCELLANEOUS RUN-TIME LIBRARY ROUTINES 57–1

57.1 Overview 57–1
57.1.1 Goals and Requirements 57–1
57.1.2 Low-Level Math Routines 57–2
57.1.3 Common Multithread Architecture Routines 57–3
57.1.4 Print System Model Client Routines 57–4
57.1.5 Open Issues 57–4

CHAPTER 58 APPLICATION RUN-TIME UTILITY SERVICES 58–1

58.1 Overview 58–1
58.1.1 Goals and Requirements 58–1
58.1.2 ARUS Routines 58–2
58.1.2.1 User Mode Virtual Memory Allocation/Deallocation Routines 58–2
58.1.2.2 Condition Handling Routines 58–2
58.1.2.3 Date and Time Conversion Routines 58–3
58.1.2.4 String Mapping Routines 58–4
58.1.2.5 Process Information Routines 58–5
58.1.2.6 Command Language Interpreter Interface Routines 58–5
58.1.2.7 Data Conversion Routines 58–5
58.1.2.8 Text String and Message Formatting Routines 58–5
58.1.2.9 String Routines 58–5
58.1.2.10 Table-Driven Parsing Routines 58–5
58.1.2.11 Math Routines 58–6
58.1.3 Open Issues 58–6

GLOSSARY Glossary–1

EXAMPLES

14–1 Prototype Object Creation Routine 14–4
14–2 Prototype Get/Set Information Object Service Routines 14–5
14–3 Sample System Service Definition 14–6
50–1 Typical Glacier Program Development 50–6

FIGURES

1–1 Glacier Client/Server Model 1–4
3–1 Mica Status 3–2
3–2 lib$status_value 3–3
6–1 Complex UJPT Tree 6–4
7–1 Virtual Address Space Layout 7–3
8–1 Overview of Mica's I/O Architecture 8–3
9–1 Dispatching System Services 9–2
15–1 Direct Access Mass Storage Function Processors and Clients 15–2
15–2 Sample Stripe Set 15–4
15–3 I/O Structure Layout for a Shadowed and Striped Files–11 Volume 15–6
15–4 I/O Request Packets Used to Satisfy the Sample Request 15–7
16–1 Magnetic Tape Function Processors 16–2
17–1 SCS Function Processor in the I/O System 17–2
19–1 How the NI Function Processor is Implemented 19–4
19–2 Mapping of Transmit Buffers to Actual Packet 19–5
21–1 How Threads Read and Write through Message FPUs 21–2
22–1 Layout of PDM and Diagnostic Subprocesses 22–3
24–1 Location of DFFP Layer in the I/O System 24–2
25–1 Relationship of Files–11 Data Structures 25–3
29–1 Object Module and Image File Format 29–3
32–1 System Management Components 32–2
33–1 Relationship of OPCOM Components 33–2
34–1 Configuration Manager Design 34–4
35–1 Read-Only System Volume Area 35–1
35–2 Read/Write System Volume Area 35–2
37–1 Interface Hierarchy of USE 37–2
38–1 Glacier Communications 38–2
38–2 Cheyenne Communications 38–3
38–3 Software Components of the Network 38–5
39–1 The Components of DECnet–Mica 39–3
39–2 Relationship of Ports and Channels in a Virtual Circuit 39–4
41–1 Overview of DECnet–Mica Network Management and Event Logging 41–2
41–2 Details of DECnet–Mica Network Management and Event Logging 41–3
42–1 Source Queue States 42–4
42–2 Sink Queue States 42–5
43–1 DFS Clients and Servers in a Network 43–2
43–2 Mica DFS System 43–3
48–1 Highly Available Cheyenne Configuration 48–2
48–2 Software Layering 48–4
48–3 Client-Resident Cheyenne Communication Components 48–12

48–4 Cheyenne Remote System Management 48–13
49–1 Transaction Services Block Diagram 49–3
51–1 Mica Compute Server Support 51–3
53–1 ULTRIX Compute Server Support 53–3
54–1 The RPC Architecture Model 54–2
55–1 The Flow of a Remote Procedure Call 55–2
56–1 DECwindows Components 56–2

TABLES

5–1 Object Architecture: Terms and Definitions 5–2
33–1 Client System Management Interface Commands 33–3
42–1 Message Queue States 42–3
42–2 Source Queue State Transitions 42–4
42–3 Sink Queue State Transitions 42–4
57–1 Low-Level Math Routines 57–3
57–2 CMA Routines 57–4
58–1 High-Level Math Routines 58–6

General

This set of chapters covers general topics relating to the Mica operating system.

CHAPTER 1

INTRODUCTION TO MICA

1.1 Overview

Mica is DIGITAL's proprietary operating system for the 1990s and beyond, targeted initially for the PRISM architecture.

The most important aspect of the design of Mica is the emphasis on building a modular system base for the future development of all types of software products. Mica is explicitly designed to break the "product P cannot ship until feature F is in the operating system" cycle, by allowing most new functionality to be added without modifying the base system. The modular design features and strong base-system functionality of Mica are summarized in this Introduction.

Mica has also become a vehicle for work in distributed system design. The two initial FRS products are the Cheyenne database server and the Glacier compute server. Both exploit client/server models, with interactive front-end processing being handled by a potentially large number of client systems, and high-performance compute- and/or I/O-intensive processing handled by a Mica-based back-end server.

The entire design of Mica is rich in innovative design concepts, with its object-based executive, layered I/O system, protected subsystems, and the exploitation of a thread-based architecture, to name just a few points. In addition, the system is written in a high-level language, to enhance its maintainability, extensibility and portability to future hardware architectures. Rounding off the base functionality of Mica is a set of superior compilers that exploit the underlying hardware and software functionality. This includes a FORTRAN compiler supporting automatic vectorization and decomposition.

The following subsection describes what makes Mica ideal as an expandable base system. The second subsection describes the process structure of Mica. The last two subsections briefly describe the initial Mica-based FRS products.

1.1.1 Expandable Base-System Functionality

The following subsections present the modular design features of Mica, which allow it to be expanded at four different levels.

1.1.1.1 Object Architecture

Fundamental to the design of Mica is the object architecture. Objects are abstract entities provided by the system that require a data structure to represent. Objects are organized into types defined in terms of the operations that may be performed on them. These operations are implemented by a collection of procedures referred to as object service routines. A few examples of objects are events, sections, processes, I/O channels, device units, volumes, open files, and timers.

The most important aspect of the object architecture is that it provides a single mechanism for controlling the identification, naming, visibility, and security of all system entities. There is a single identification mechanism for all objects in the system, based on a 64-bit object ID. Optionally, any object may also have an ASCII name, which may be translated (qualified by object type) to find its ID. Objects are referenced via object IDs stored in object containers at three levels of visibility: system, job, and process. Visibility of objects may be allowed for all processes in a system, all processes in a

job, or a single process, depending on whether the object is created in a system object container, job object container or process object container, respectively.

Objects are the single focus of security in the system. Every object can optionally have an access control list (ACL), which allows protection to be specified on a per-object basis. The ACL is matched against the security profile of any thread attempting access to the object.

Therefore, at the lowest level of the system, new functionality may be added by loading new object types and object service routines into the system.

1.1.1.2 Layered I/O System

The I/O system of Mica is implemented as a layer on top of the object architecture.

The I/O architecture is designed to facilitate the successive layering of virtual support on top of actual physical devices. Each layer is implemented by an entity referred to as a function processor. Function processors implement many levels of I/O, including those implemented in VMS by device drivers, pseudo drivers, ancillary control processors (ACPs), and extended QIO processors (XQPs). Mica has additional function processors to implement such things as disk striping, disk shadowing, each of the DNA network layers, and so on.

Function processor units (FPUs) are objects created for function processors, to which channels may be created for subsequent I/O operations. For example, the Files–11 function processor has function processor units for volumes, device function processors have function processor units for devices, and so on.

Function processors may be layered by creating a channel to one FPU and referencing the channel in another FPU.

Thus, the I/O architecture provides another mechanism by which new functionality may be added to the system in a modular fashion. In fact, by implementing new function processors that support pre-existing function processor interfaces, it becomes possible to "plug in" new functionality such as a new file system or network transport.

1.1.1.3 Protected Subsystems

A protected subsystem in Mica is a process that implements a protected area of functionality in user mode. Protected subsystems accept requests via remote procedure call (RPC). The use of RPC makes it completely transparent that the protected subsystem is running as a separate process.

Key to the protected subsystem support is the passing of a security profile with the RPC from the calling thread to a thread in the protected subsystem. The protected subsystem thread can then access objects in the system according to either its caller's security profile, its own security profile, or a security profile that is a merger of the two.

Protected subsystems are the favored way of adding functionality to Mica. Since protected subsystems run as separate processes in user mode, they are more robust, easier to code and debug, and easier to maintain. They do not compromise the integrity of the base system. Protected subsystems also generalize in a straightforward manner to distributed system implementations via network RPC.

Some examples of functionality implemented as protected subsystems in Mica are the Configuration Manager, System Management, and OPCOM.

1.1.1.4 Client/Server Tools

The following components, which are part of the client/server interface for Mica, could be useful in developing other client/server systems:

- The underlying Mica network architecture, including network RPC.
- The Distributed File System (DFS), which implements file access within a distributed system.
- The server-oriented job controller, which handles remote image activation requests on a Mica server. It also facilitates binding back to the client for accessing the initiating terminal and retrieving environment information (such as logical names, defaults, and so on) from a "client context server" on the client.

\Cheyenne uses a subset of the above.\

Figure 1–1 depicts Glacier as an example of a client/server model.

There are some good reasons why future software products after Cheyenne and Glacier may also wish to use a client/server implementation. In the two initial client/server products, the user interface is removed from the powerful Moraine server system to frontends that have much less compute power. But since the user interface is mostly characterized by high I/O latency and low compute requirements, the user still receives good response time. If, on the other hand, the server implemented the user interface directly, and attempted to support hundreds of users simultaneously, the aggregrate compute requirements for the user interface could be substantial. The server is thus much better utilized and is capable of supporting more users if it is limited to work dispatched to it from clients that is more characterized by high I/O bandwidth and compute requirements.

1.1.2 Process Structure and Threads

Mica supports job and process objects, which are very similar to jobs and processes in VMS. It also supports user objects, which represent users that have been validated in the system and are allowed to have jobs. However, a very important aspect of the Mica architecture comes from the fact that Mica supports the concept of threads.

A thread is the entity of execution within a process. All threads within a process share the same address space and object containers, but they have separate stacks, registers, hardware context, exit handlers, and AST queues. A process may support multiple threads of execution in parallel; in fact, these threads may run on multiple processors at the same time.

Mica employs multiple threads to increase the level of parallelism within the operating system itself. More important, however, is the use of threads in compute-intensive user software. Compilers for Mica can generate code that uses threads to implement parallel decomposition.

Through the use of threads at both the system and application levels, Mica is inherently a parallel system. This fact further enhances Mica's suitability as a powerful software base for the future.

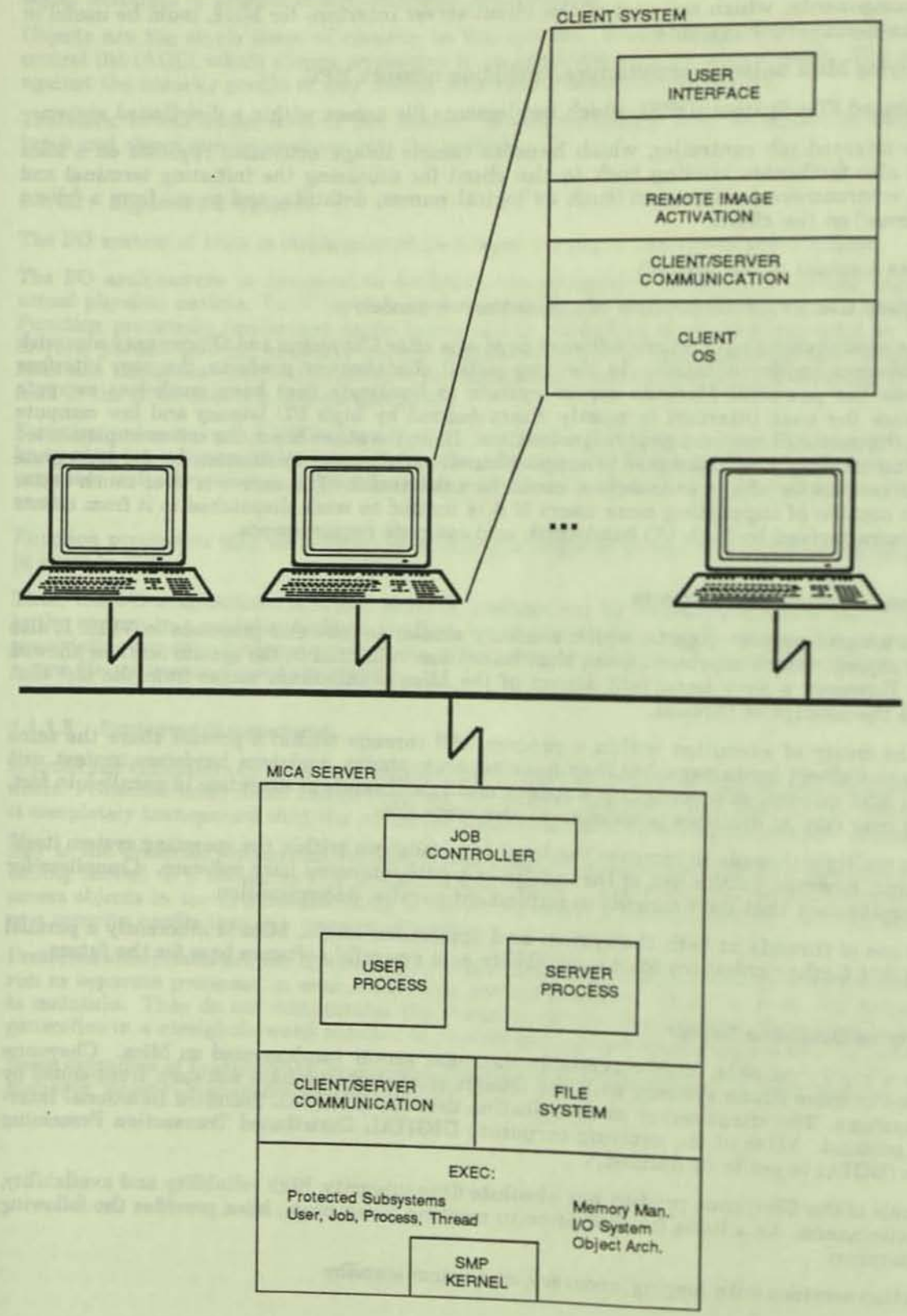
1.1.3 Cheyenne Database Server

Cheyenne is a highly reliable, highly available database server implemented on Mica. Cheyenne consists of one or more Stone systems with the Quartz relational database software, front-ended by VAX client systems. The client/server communication uses the DIGITAL Standard Relational Interface (DSRI) protocol. \Use of the evolving corporate DIGITAL Distributed Transaction Processing Architecture (DDTA) is yet to be decided.\

The main goals of the Cheyenne product are absolute data integrity, high reliability and availability, and high performance. As a basis for Cheyenne in meeting these goals, Mica provides the following important features:

- Transaction services with logging, recovery, and warm standby

Figure 1–1: Glacier Client/Server Model



- Disk shadowing
- Automatic hardware reconfiguration, including a spare strategy for shadowed disks
- An aggressive strategy for the handling of transient machine faults
- Threads for parallel processing of transactions
- A special interprocess communication mechanism for database processes

These features will also help Mica to evolve as a strong base for general purpose transaction processing.

1.1.4 Glacier Compute Server

Glacier is a high-performance compute server for scientific and other compute-intensive applications. A typical Glacier system consists of a Glacier-based Moraine SMP system with scalar-vector or scalar-only modules, front-ended by a potentially large number of workstations or VAX frontends. The client/server interface employs RPC, and is designed in such a way as to allow the support of a variety of systems as front ends.

The main features of Mica exploited in Glacier are:

- High-performance and reliable disk I/O via striping and shadowing
- The use of multiple threads for parallel processing of single-stream applications
- Large virtual and physical address space
- The Distributed File System (DFS)
- A set of superior compilers
- The Applications Integration Architecture (AIA)

An important part of the compute server concept is the philosophy of allowing multiple client systems to provide their own user interface and software development environment, complete with client-specific command languages, editors, and file utilities. This allows the user to develop compute-intensive applications on the client system in a familiar environment (initially VMS, later ULTRIX and others), and then execute these applications transparently on the server. Some compute-intensive DIGITAL software also executes on the server, such as the compilers and linker.

The server itself is designed to support portable applications that do not specify system-specific calls. This is made possible via the use of state-of-the-art standard compilers (such as FORTRAN, C, Ada, and Pascal) and the Applications Integration Architecture (AIA).

1.2 WDD Structure

Following is a complete list of the WDD chapter overviews, in the order they appear in the Mica Overview Document. They are grouped into logical areas, which are named in the following list in all uppercase. The chapters within each area are indented below their corresponding area title.

GENERAL
- Introduction to Mica
- Naming Standards and Pillar Coding Style Guidelines
- Status Values, Messages, and Text Formatting

EXECUTIVE
- The Kernel
- Object Architecture
- Process Structure
- Memory Management

I/O Architecture
System Service Architecture
Security and Privileges
Condition, Exit, and AST Handling
Booting
System Dump Analyzer and System Debugger

EXECUTIVE ROUTINES
Executive Routines

I/O
Direct Access Mass Storage Function Processors
Magnetic Tape Function Processors
System Communication Services
XCA Function Processor
NI Function Processor
Console Support
Message Function Processor
PRISM Diagnostic Monitor
Error Logging

FILE SYSTEM
Disk File System Function Processors
Files–11 ODS2 Function Processor
Record Management Services
Caching
File Management Utilities

IMAGE RELATED
Object Module and Image File Format
Linker
Image Activation

SYSTEM MANAGMENT and ADMINISTRATION
System Management
Operator Communications
Configuration Management Software
System Volume Layout and Software Installation

TESTING and PERFORMANCE MEASUREMENT
Performance Monitor
User-Level System Exerciser

NETWORK
Mica Network Overview
Network Services
DNA Naming Service Clerk
DECnet Startup, Shutdown, Management, and Logging
Quartz Interprocess Communication

DISTRIBUTED FILE SERVICES
Distributed File Service Introduction
Distributed File Service Management
Distributed File Service Communication Function Processor
Distributed File Service Client Function Processor
Distributed File Service Server

DATABASE SERVER
Cheyenne Overview
Transaction Services

COMPUTE SERVER
- Glacier Overview
- Mica Compute Server Support
- VMS Compute Server Support
- ULTRIX Compute Server Support
- Protected Subsystems and RPC
- RPC Stub Generator
- AIA User Interface
- Miscellaneous Run-Time Library Routines
- Application Run-Time Utility Services

CHAPTER 2

NAMING STANDARDS AND PILLAR CODING STYLE GUIDELINES

2.1 Overview

This chapter defines the naming conventions to be used for names accessible from user-mode programs throughout Mica. It also presents coding style guidelines for all code written in *Pillar*, the software development language for the Mica operating system.

2.1.1 Goals

The overall goal in setting naming standards and coding guidelines is to ensure consistency across code written for the Mica operating system. Specifically:

- Naming standards:
  - — Present a consistent, easy-to-remember name space to users and developers
  - — Aid future developers in maintaining and extending the software
  - — Ensure that customer-written software is not invalidated by future releases of DIGITAL products that add new names
- Pillar coding style guidelines:
  - — Ensure that Mica system software written in Pillar is more maintainable and extensible
  - — Improve the overall presentation of Pillar code examples in Mica documentation

2.1.2 Naming Standards

Naming standards will be used to define names for all public software interfaces for layered products and bundled Mica software. The Naming Standards and Pillar Coding Style Guidelines chapter provides conventions for:

- Facility names
- Module names
- System service and system routine names
- Procedure parameter names
- Mica system files and directory names
- Named types
- Global variables
- Compile-time named constants
- Message names
- Item codes used in item lists

- System and group logical names used to alter, define, or control a facility
- Compile-time facility macros and procedures

2.1.3 Pillar Coding Style Guidelines

Pillar coding guidelines will be used by all Mica software developers to promote code consistency across all Mica system software. The Naming Standards and Pillar Coding Style Guidelines chapter provides conventions for:

- Statement indentation policy
- Capitalization policy
- Source line length
- Order of declarations
- Format and policy of multi-line statements and multi-statement lines
- Block and line comment format
- Use of "whitespace" to improve code readability
- Pillar statement formats including:
  - Executable statements such as IF/THEN/ELSE, CASE, and LOOP
  - Record, enumerated type, and set declaration formats
  - Procedure declarations (external and procedure completions)
  - Procedure invocations
- Module and procedure layout format

CHAPTER 3

STATUS VALUES, MESSAGES, AND TEXT FORMATTING

3.1 Overview

A *status value* passes information regarding the success or failure of a process, thread, I/O service, or procedure back to the thread which created or called it. Status values are also used to organize and index messages that convey information about status values in textual form.

This chapter:

- Defines the format of Mica status, the data structure which contains a status value.
- Defines the format of status values.
- Describes the mechanisms used to translate status values in text strings.
- Describes the organization of messages and message files.
- Describes the use of messages and message files for internationalizing text.
- Outlines the text formatting support provided on Mica. While such support is an important part of message access and display, it is general purpose in nature and may be used in any programming situation where text formatting is required.

3.1.1 Goals

The primary goal of this implementation is to provide a consistent, easy-to-understand, and easy-to-use way of organizing definition of and access to status information, message text, or both. Within this general goal are the following specific goals:

- To provide a local message capability which allows message definition and access without the requirement of facility registration
- To provide a convenient way of separating text from an image that uses it, and to allow the text to be rewritten in another natural language without affecting the image
- To describe and encourage the use of the message capabilities for all user-displayed text in a program, not just status messages, as a way to internationalize programs more easily
- To provide a text formatting capability that addresses internationalization requirements

3.1.2 Status on Mica

Mica status is 64 bits. The first 32 bits are the status value; depending on the type of status, the second 32 bits may or may not be used. Mica defines three types of status: *facility-registered status*, *local status*, and *internal status*. The format of each type is shown in Figure 3–1.

- Facility-registered status—The status value contains a number indicating which facility generated the status. The second 32 bits are not used.
- Local status—The status value has the local status bit set. A full 25 bits are used for the message number as a facility number is not required. The second 32 bits of the status contain the address of a message data structure used to acquire the message text.
- Internal status—This type of status is used internally by a particular facility. The first 32 bits is a facility-registered status value. The second 32 bits may be used in whatever way the facility desires. An internal status normally does not appear outside the facility that uses it because outside the facility, the second 32 bits of the status are ignored.

Figure 3–1: Mica Status

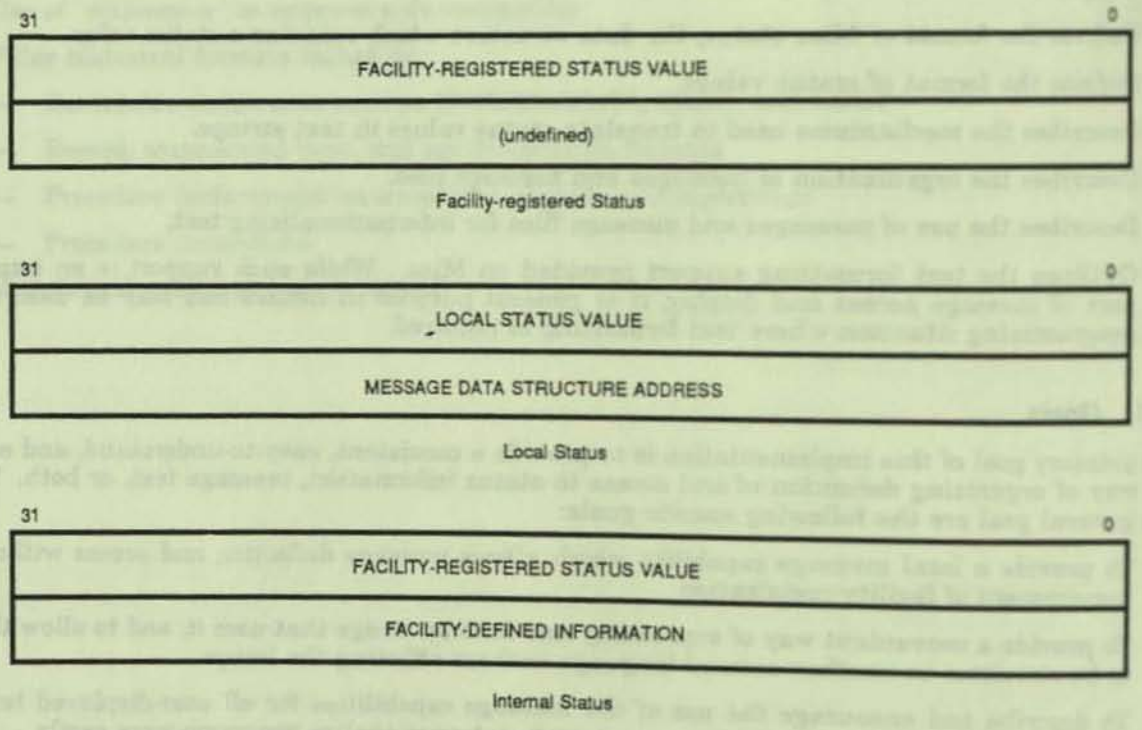


FIG0

Pillar's predefined data type STATUS is 64 bits. For languages which do not support 64 bit return values, such as C and FORTRAN, procedures which return local status should include an optional argument to the procedure which returns a condition record. This condition record contains the full 64 bit status, all of which are required for local message retrieval.

3.1.3 Status Values

There are two kinds of status values: *facility-registered status values*, and *local status values*. Status values are longword values used to:

- Indicate the exit status of a process
- Indicate the exit status of a thread
- Return status from a remote procedure call
- Return completion status from an I/O request
- Return status from a procedure or function call (such as a run-time library function)
- Organize local messages, that is, internal messages within a program

Additionally, values in status value format are used to organize and access nonmessage text local to a facility.

Status values have the binary format shown in Figure 3–2:

Figure 3–2: lib$status_value

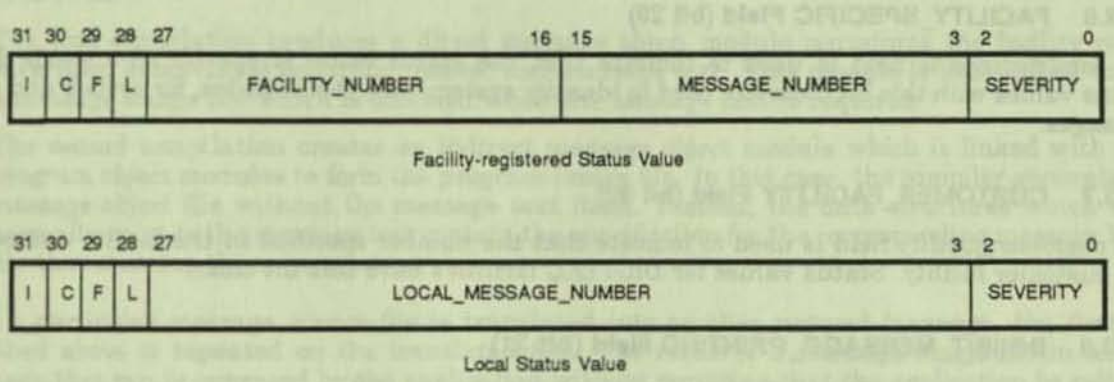


FIG1

The sections below describe each field of a status value.

3.1.3.1 SEVERITY Field (bits <2:0>)

The *severity* field of a status value indicates the basic success or failure of the producer of the status. Severity is represented as a binary value in the range 0 to 4 (values in the range of 5 to 7 are reserved to DIGITAL).

Successful completion is indicated by an odd-valued severity. Even severity values indicate partial or complete failure.

3.1.3.2 MESSAGE_NUMBER Field (bits <15:3>)

The *message_number* field of a status value is used to identify which of a set of several possible conditions this status value represents. The message routines use this value to index into a message section to obtain the corresponding message text. This field is defined only for facility-registered status values.

3.1.3.3 FACILITY_NUMBER Field (bits <27:16>)

The *facility_number* field of a status value is used to identify the producer of the status value. Each facility must have its own unique facility number. This field is defined only for facility-registered status values.

The facility number 0 is reserved for system-wide status values. The facility name corresponding to facility number 0 is STATUS.

3.1.3.4 LOCAL_MESSAGE_NUMBER Field (bits <27:3>)

The *local_message_number* field of a status value is used to index into a message section to obtain message text for a local message. This field is defined only for local status values.

3.1.3.5 LOCAL_STATUS Field (bit 28)

The *local_status* field is used to indicate that the status value is local. Local status values are used to organize facility local messages without the requirement of facility registration. Local status values have this bit set; facility-registered status values have this bit clear.

3.1.3.6 FACILITY_SPECIFIC Field (bit 29)

The *facility_specific* field is used to indicate that the status value is specific to a single facility. Status values with this bit clear are used to identify system-wide status codes, for system and shared messages.

3.1.3.7 CUSTOMER_FACILITY Field (bit 30)

The *customer_facility* field is used to indicate that the number specified in the facility number field is a customer facility. Status values for DIGITAL facilities have this bit clear.

3.1.3.8 INHIBIT_MESSAGE_PRINTING Field (bit 31)

The *inhibit_message_printing* field is used to inhibit display of the message by message output routines. This bit is set by system routines that display the resulting message, so that the message is not displayed twice.

3.1.4 Status and Text Messages

Status messages are text strings used to describe a status value to a user in a natural language. A complete status message consists of:

- Facility name—A short string of characters indicating the facility to which the status is registered.
- Severity—A single letter indication corresponding to the severity of the status.
- Abbreviated condition name—A short string of characters identifying the status in an abbreviated manner.
- Message text—A string of characters describing the status in detail, possibly with formatted parameters specific to the error occurrence.

Text messages are text strings used to provide non status related information to a user in a natural language. A text message is the same as the message text portion of a status message.

3.1.4.1 Status Message Format

By default, status messages are assembled in the following format:

```
%FACILITY-S-ACONDNAME, message text
```

"FACILITY" is the facility name, "S" is the severity, and "ACONDNAME" is the abbreviated condition name. A user or facility may request that certain parts of a status message be excluded when the message is assembled. The default message format may be changed with a CLI command (such as SET MESSAGE for DCL). A logical name is used to convey the current message format setting between a CLI running on a client system and a program running on the server.

3.1.4.2 Message Source Files and Compilation

Messages are created in text format using a text editor. A file consisting of a collection of facility name, abbreviated condition names, severity condition values, and message text is called a message source file. Message compilation is the process of creating a message object file from a message source file. Mica provides message compilation capabilities as part of the Pillar compiler.

The message compilation facility provides a way to internationalize messages by allowing the message text and formatting information to be separated from the image file. The message source file is compiled twice:

1. The first compilation produces a direct message object module containing the facility names, severities, abbreviated condition names, and message text. This module is then linked to form a message image file which is accessed when the message text is required.
2. The second compilation creates an indirect message object module which is linked with other program object modules to form the program image file. In this case, the compiler generates the message object file without the message text itself. Instead, the data structures which would normally point to the message text contain the specification for the corresponding message image file that contains the message text.

Once a particular message source file is translated into another natural language, the first step described above is repeated on the translated file. The result is a message image file in another language that can be accessed by the application without requiring that the application be relinked.

3.1.4.3 In-Memory Message Organization

Mica status value and message support stores message information in a *message section*. Message sections are pointed to by message section descriptors. There are two types of message section descriptors: the *direct message section descriptor* and the *indirect message section descriptor*.

- Direct message section descriptors contain a pointer to a message section that is generated at the time the message file is compiled. This is the case when the entire message section (including text) is linked with the image.
- Indirect message section descriptors contain a pointer to the name of the file containing the message section and a null message section pointer. When the message section is first accessed, the message image file containing the corresponding message section is read into memory and the pointer to the message section is updated. This is the case when the two-step process described in Section 3.1.4.2 is used to create internationalizable messages.

Message support also allows message sections to be chained together. This allows multiple language versions of a given message section to be available at the same time. Such support is required for multithreaded server processes whose clients may have different default languages.

3.1.4.4 Accessing and Displaying Messages

Mica provides three routines for accessing and displaying messages:

- The *lib$get_message* routine is used to obtain and format status messages. This routine takes a condition record as input and obtains and formats the status message corresponding to the status value in the condition record.
- The *lib$display_message* routine is used to obtain, format, and display status messages. This routine takes a condition array as input and obtains and formats the status messages corresponding to each condition record's status value.
- The *lib$get_text* routine is used to obtain and format a message corresponding to a supplied local status value and message section.

Translation of a status value to a message depends on the type of status:

- Facility-registered status values—Translation is accomplished by searching one or more message sections. Each process has access to two groups of message sections; image and system. Image message sections are those loaded with the image. System message sections are those shared across the entire system. The translation routines search image message sections first, followed by system message sections.
- Local status values—Translation is accomplished by searching the specified message section. If the specified message section does not contain the index specified by the *local_message_number* field, the translation fails. No other message sections are searched.

3.1.5 Text Formatting

Mica status and message support also includes text formatting capabilities. The *lib$format_text* routine provides support for a new set of formatting directives. Specific goals for this functionality are:

- To move data type and access information out of the formatting control string, placing it with the arguments instead
- To provide full parameter positioning and formatting capabilities required for full internationalization support

The directives provide:

- Formatting information such as width, radix, and fill
- Positioning information that allows parameters to be positioned differently for different natural languages
- Special formatting requests such as system date and time
- A means of specifying that directives are to be repeated in a controlled fashion

3.1.6 Open Issues

- Message compilation support will be provided by the Pillar Compiler Group. Development of the Pillar Message Compiler will not occur until late calendar 1988. The text formatting language and message data structures may have to be modified as the message compiler is developed.

Executive

This set of chapters describes the components of the Mica executive.

CHAPTER 4

THE KERNEL

4.1 Overview

4.1.1 Requirements

The kernel is the lowest layer of software in the system and, as such, is positioned closest to the actual hardware. The kernel is a single layer of code that must implement all interprocessor synchronization, thread dispatching, exception handling, and fork processing. It must also keep the system time and provide services to device drivers for handling interrupts.

The kernel presents a formal interface to the next higher level of software (the executive) that is free of the problems associated with synchronizing various activities on multiple processors and which automatically implements symmetrical multiprocessing (SMP) capabilities.

The kernel attempts to implement no policy. That is the province of the higher levels of software in the system. There are, however, some algorithms that must be implemented in the kernel for efficiency, and therefore, some policy will be included in the kernel. Such a case is the way in which the priority of a thread decays over time. For those cases where it is essential for policy to be located in the kernel, external controls will be provided so that executive software can influence, if not directly control, the actions of the kernel.

4.1.2 Functional Description

4.1.2.1 Environment of the Kernel

The kernel runs in kernel mode, usually at an interrupt priority level (IPL) of 2. This is the priority level at which dispatching occurs. The kernel can be executed simultaneously on all processors in a multiprocessor configuration, and synchronizes access to critical regions as appropriate.

Software within the kernel is not context switchable, whereas all software outside the kernel is always context switchable. In general, executive software is not allowed to raise IPL above 1, or otherwise block context switching, and must use kernel procedures to synchronize its activities.

The kernel is not pageable and cannot take page faults.

All software outside the kernel is written in Pillar. Kernel software is a mixture of Pillar and assembly language. All interfaces to the kernel are defined in Pillar and exported to other programs. It is expected that the size of the kernel will be approximately 8K instructions.

4.1.2.2 Interaction With the Executive

Executive software also runs in kernel mode. It implements system services, memory management, user-level object support, the file system, network access, and device drivers; and it sets system policy.

Executive software communicates with the kernel via a set of data abstractions called kernel objects, and a set of operations that can be performed on these objects. Kernel objects are referred to by address and should not be confused with user objects as defined by the object architecture. Kernel objects are not accessible to user software. An example of a kernel object is an event, which provides a form of synchronization.

There is no firewall protection provided between the kernel and executive software. They both run in kernel mode and can potentially disrupt each other's activity. There is, however, a formal interface between executive software and the kernel, and a well-defined set of rules that must be obeyed.

Normally, the kernel does little or no checking of procedure arguments supplied by the executive; however, debugging software can be conditionally compiled into the kernel to ensure the correctness of calls to kernel procedures. For those cases that the kernel does check argument values for consistency, an error condition is raised via the standard condition mechanism when a parameter value is found to be in error.

4.1.2.3 Primary Kernel Data Structures

The following are the primary data structures defined and used by the kernel:

- The system control block (SCB)—The SCB is an architecturally defined structure that contains an array of exception and interrupt service routine addresses used to service interrupts and exception conditions. The base address of the SCB is stored in the system control block base register (SCBB).
- The processor control block (PB)—The PB contains a collection of processor-specific information. Examples of the information contained in the PB include a pointer to the thread object of the current thread, the processor-specific fork queue header, and counts of the interprocessor interrupts that have occurred. The address of the PB is stored in the processor base register (PRBR), which is defined by the PRISM architecture.
- An array of pointers to the PBs—There is a pointer to the PB for each processor in the system. The index into this array for each processor is stored in the WHAMI register for the processor.
- Spin locks—Spin locks are used to achieve multiprocessor synchronization. In the kernel, spin locks are used to synchronize access to eight kinds of entities:
  1. Dispatcher database
  2. Power-up request queue
  3. Power-up status queue
  4. VAX port queues
  5. Device work queues
  6. Active I/O interrupts
  7. Processor request
  8. Kernel debugger
- Kernel objects—Kernel objects are data abstractions that are necessary to control processor execution and synchronization. Kernel objects parallel objects as defined in Chapter 5, Object Architecture, but are not directly available to user software and are addressed by pointers rather than object IDs. Kernel objects are divided into two categories, dispatcher objects and control objects. The kernel objects are:

Dispatcher objects	Control objects
Event	AST
Mutex	Device work queue
Queue	Interrupt
Semaphore	Power-up request
Timer	Power-up status
Thread	Process
	VAX port queue

- Dispatcher database—The dispatcher database is used when choosing which threads should be active at any point in time. The database is a collection of data structures that contains information such as a list of threads ready for execution, and a record of which processors are executing threads at which priority levels.
- Timer queue—The timer queue is a binary tree of timer objects that are each set to expire at a specified time.
- Power-up request and status queues—The power-up request and status queues are used to notify threads when a power recovery interrupt is received by PRISM hardware.
- Performance data—The kernel collects and stores performance data in various private data structures.

4.1.2.4 Primary Kernel Functions

The primary functions of the kernel include:

- Multiprocessor coordination—To coordinate the activity of multiple processors the kernel uses spin locks for synchronization and interprocessor interrupts for notifying other processors of work to be done. Executive code outside the kernel can use either spin locks or mutex objects to implement mutual exclusion.
- Thread dispatching—The kernel supports 64 levels of thread priority. The highest 16 levels are referred to as real-time priorities and the lowest 48 levels as class priorities. The kernel implements dispatching, which chooses exactly which thread to execute next. Scheduling, which selects the threads that are eligible for execution, is the province of higher levels of software.
- AST Processing—The kernel provides services for queuing and delivering asynchronous system traps (ASTs) to target threads. A combination of software state and hardware registers is used to determine the correct time to interrupt thread execution.
- Interval timer support and the system time—The interval timer is used by the kernel for maintaining the system time, accumulating accounting and performance information, updating thread quantums, and timer queue maintenance. The system time is maintained as a quadword count of 100ns intervals and is initialized to zero when the system is booted.
- Address space number (ASN) Management—The kernel provides for complete management of the assignment of address space numbers (ASNs). ASNs are used to tag translation buffer entries and therefore avoid flushing at every context switch.
- Powerfail Recovery—Powerfail recovery support is provided by the kernel via power-up request and status objects. In conjunction with raising IPL, these objects provide a driver thread with the capability to interrupt its execution and/or have a status variable set when a power recovery interrupt is received by PRISM hardware. Power-up status objects may only be used directly by kernel-mode code. Power-up request objects are intended primarily for use by driver threads, but can also be provided to user-mode programs via executive objects.

4.1.2.5 Performance Data Collection

The kernel collects various categories of performance data during its execution so that both the designers and users of the system can analyze and improve its performance. The data structures required to record this data are private to the kernel and, therefore, are not directly accessible to executive software. Executive software can retrieve the following data, however, by calling a kernel procedure that returns the desired category of data:

- Number of currently computable and waiting threads
- Processor fork queue depth
- Context switch headway
- Number of interprocessor interrupts (for each kind of request)
- Interrupt data for an interrupt vector
- Contention data for device work queues and mutexes
- Processor mode data
- Dispatcher object wait queue depth

CHAPTER 5

OBJECT ARCHITECTURE

5.1 Overview

5.1.1 Introduction

This chapter describes the software architecture of objects. It describes what objects are, and defines the data structures and operations necessary to support objects.

5.1.2 What Is an Object?

Objects are abstract elements provided by an operating system that may be accessed by a user or a program. Typically, objects are defined in terms of the operations that may be performed upon them (for example, create, clear, set, get information, wait, delete) and their relationships to other objects. The reason for categorizing these elements as objects is to provide a single, standardized set of rules for creating, naming, protecting, accessing, and managing them. For example, each object has a unique ID value (called an object ID) which may be used to identify it. Objects at the job and process levels are only directly expressible by threads in that job or process.

5.1.3 Scope

It is important to understand that this chapter only defines the architecture of objects, not all object types. It is necessary that some objects or parts of objects be defined as part of this architecture.

5.1.4 Requirements and Goals

- Software development goals
  - Provide an extensible, yet rigorous framework for the definition and manipulation of executive-controlled data structures.
  - Maintain management consistency. The management of objects, in terms of actions taken to fulfill service requests, should be as object-type independent as possible. For example, standard routines and procedures can be established for determining whether access to an object should be granted.
  - Provide new object definition support. It should be possible to add new object types to the system without having to modify existing system code. This means that the interface between the kernel/executive system software and objects must be well-defined, and that the kernel/executive need not have knowledge of the internals of all objects.
- Interface goals
  - Provide consistent specification. The ways in which each object in the system may be specified by users should be minimized and kept consistent with the manner in which other objects are referenced.

— Provide consistent operations. There are some operations that apply to a set of objects within the system, such as wait. The definition of what these operations mean to each object should be kept simple and similar to their definition for other objects.

— Support level independence. Where possible, the operations that may be performed on an object type, and the behavior of that object, should not be dependent upon the level (system, job, or process) at which that object has been created. This allows applications to be developed in the relative safety of process and job levels before being moved to a more shareable level, with minimal change in behavior.

— Provide security and protection. The method of determining which objects a user may refer to, and which operations may be performed on those objects, should be the same for all objects. This is the basis for Mica security.

5.1.5 Functional Description

The object architecture runs in kernel mode at IPL 0. Through the use of mutexes, object architecture procedures can simultaneously execute on multiple processors.

The object architecture provides a framework for creating object-specific services. These services include creating, deleting, allocating, referencing, name translating, and getting information about objects. For example, the object service to create an event, and the object service to create a thread both invoke the same object architecture-defined routine to create the object.

The object architecture provides a hierarchical visibility structure for objects. When an object is created, it is placed at one of three levels: system, job, or process. Objects at the system level are visible to all threads on the system. Objects at the job level for a particular job are only visible to threads in that job. Objects at the process level for a particular process are only visible to threads in that process. For example, a thread cannot access an object that is at the process level for another process, because it cannot express an object ID for that object.

Each level can contain one or more object containers to catalog objects at that level. There are two types of object containers at the process level: Process-private object containers and display object containers. Objects stored in a process-private object container are only visible to the process with which the container is associated. Objects stored in a display object container are visible to the associated process, and any of its descendant processes.

Objects are referred to by object ID. If a program refers to an object name, this name must be translated to an object ID. An object name is unique within a container for each object type at each processor mode. When a user attempts to refer to an object using an object ID, the user's access rights are compared to the access control information associated with an object. If there is a match, the user may access the object.

An object may be allocated to a user, identifier ID, job, process, or thread. This allows objects to be shared among restrictive classes of users.

An object ID is deleted when the object container holding the corresponding object is deleted, or when the object ID is explicitly deleted. The object itself, however, is not deleted until the ID is deleted, and there are no outstanding references to the object.

Table 5–1 summarizes key object architecture terms and components.

Table 5–1: Object Architecture: Terms and Definitions

Object Identification and Names		
Term	Type	Definition
Object ID	64-bit Value	Used to refer to an object.

Table 5–1 (Cont.): Object Architecture: Terms and Definitions

Object Identification and Names

Term	Type	Definition
Principal Object ID	Object ID	Associated with an object at object creation. An object has exactly one principal object ID.
Reference Object ID	Object ID	Optionally associated with an object. An object may have zero, one, or more reference IDs.
Object Name	Character String	Together with type and mode, an object name can be translated to an object ID. The combination of object type, mode, and name string is unique within a single object container.
Object Name Table	Data Structure	Tracks object names within an object container. When an object container is created, a name table is also allocated, and that address is stored in the object container's body.

Object Hierarchy

Term	Definition	Description
Object Container	Object Type	Objects of this type contain pointers to other objects. They are used to organize large numbers of objects.
Object Level	–	Indicates the scope of visibility of an object container.
System Level	Object Level	Objects at this level are potentially accessible to all processes on the system.
Job Level	Object Level	Objects at this level are potentially accessible to all processes in a given job.
Process Level	Object Level	Objects at this level are potentially accessible to all threads in a given process. Containers at this level can be ether display or private.
Display Container	Object Container	Objects in such containers are accessible to a given process and all of its descendants.
Private Container	Object Container	Objects in such containers are accessible only to a given process, and not to its descendants.
Container Directory	Data Structure	Used to organize large numbers of containers. All threads have the same system container directory. All threads in a job have the same job container directory. All threads in a process have the same process container directory.
Object Header	Data Structure	Fixed-format data structure that contains object type-independent data. This header is used by the executive without necessarily knowing the type of object it is accessing.
Object Body	Data Structure	A data structure that is specific to an object type.

Table 5–1 (Cont.): Object Architecture: Terms and Definitions

Object Type		
Term	Definition	Description
Object Type	–	Object type determines what operations can be performed on an object.
Object Type Descriptor (OTD)	Data Structure	Describes what operations are supported for what object types. There is one OTD for each object type.

Miscellaneous		
Term	Definition	Description
Object Service Routines	System Routines	Implement operations that can be performed on objects. Some object service routines are particular to a certain type of object; others are supported across all object types.
Object Allocation Block	Data Structure	Contains information about object allocation.

5.1.6 Object-Related Operations

The following types of operations can be performed on most types of objects:

- Creating an object
- Protecting an object
- Translating an object ID
- Deleting an object
- Creating references to an object
- Making a temporary object
- Marking a new object as temporary
- Allocating an object
- Deallocating an object
- Getting information about an object
- Changing the name of an object

CHAPTER 6

PROCESS STRUCTURE

6.1 Overview

This paper describes the components of the Process Chapter for the working design document of the MICA system. The Process Chapter describes the architecture of the User, Job, Process, Thread (UJPT) hierarchy in terms of its external interfaces and data structures. The chapter also describes the UJPT implementation in terms of its algorithms and dependencies on other portions of the MICA system (e.g. the kernel and object architecture).

6.1.1 Goals/Requirements

The goal of the UJPT architecture is to provide a vehicle for controlling multiple threads of execution in a single address space. The architecture provides facilities for resource usage control, security profile management, address space and image management, and object container directory services.

6.1.2 UJPT Hierarchy

The UJPT architecture consists of a hierarchy of objects. The objects provide a logical grouping of functionality and control.

6.1.2.1 The User Object

The User object appears at the highest level of the UJPT hierarchy. Its primary function is to provide a focal point for acquiring security profiles and resource quotas/limits for its underlying objects.

The User object is implemented as a system level object in the "USER$OBJECT_CONTAINER" object container.

6.1.2.1.1 Functional Interface

The MICA executive provides entry points capable of setting and extracting various attributes of a User object. An entry point also exists to delete, or "force exit", a particular User object which simple "force exits" all jobs of that user. There is no user interface for creating User objects since they are only created as a side effect of creating a Job object.

6.1.2.2 The Job Object

The Job object appears at the second level of the UJPT hierarchy. Its sole function is to provide a set of resource limits for a collection of processes running together as a job. The job object also provides a job level object container directory.

The Job object is implemented as a system level object in the "JOB$OBJECT_CONTAINER" object container.

6.1.2.2.1 Functional Interface

The MICA executive provides entry points capable of creating and deleting job objects, and setting and extracting various attributes of a Job object. The creation of a the first job for a MICA user causes the user object for that user to be created.

As part of Job object creation all of the necessary support data structures are created including a job level object container directory and, the associated kernel mutex.

6.1.2.3 The Process Object

The Process object appears at the third level of the UJPT hierarchy. Its primary function is to provide an address space and a program image for a set of threads. The Process object is the target of all accounting information. The Process object can also act as a focal point for control operations.

There can be multiple processes in a job. Processes created as a result of job creation are *top level* processes. Once established, a process may cause the creation of other processes. These new processes are *sub–processes* ,or *child processes*. The refer to the creating process as their *parent process*.

The Process object is implemented as a system level object in the "PROCESS$OBJECT_CONTAINER" object container.

6.1.2.3.1 Functional Interface

The MICA executive provides entry points capable of creating and deleting Process objects, setting and extracting various attributes of a Process object, and performing control operations on all threads of the process.

Control operations are Suspend/Resume Process, Hibernate/Wake Process, and Signal Process. Control operations performed on Process simply perform the operation on all threads of the process.

As part of Process object creation all of the necessary support data structures are created including the read only process control region (PCR), and a process level object container directory. The PCR is part of the processes user–mode read only address space. The MICA executive places information in the PCR so that the process can read it without entering the system.

6.1.2.4 The Thread Object

The Thread object appears at the lowest level of the UJPT hierarchy. Its primary function is to provide a thread of execution. The Thread object is the schedulable entity in the MICA system. It maintains the processor state as it executes the program steps of an image. The Thread object is the consumer of resources, but the accounting for these resources occurs in the Process object. The Thread object can also act as a focal point for control operations.

The Thread object is implemented as a process level object in the "THREAD$OBJECT_CONTAINER" object container.

\ I think threads should be system level objects ?\

6.1.2.4.1 Functional Interface

The MICA executive provides entry points capable of creating and deleting Thread objects, setting and extracting various attributes of a Thread object, and controlling Thread objects.

Thread object control services are Suspend/Resume Thread, Hibernate/Wake Thread, and Signal Thread.

As part of Thread object creation all of the necessary support data structures are created including the read only thread control region (TCR), the read/write thread environment block (TEB), and user and kernel stacks. The TCR is part of the processes user–mode read only address space. The MICA executive places information in the TCR so that the thread can read it without entering the system.

The TEB is part of the user–mode thread architecture. The MICA executive initializes the TEB to point to the TCR.

6.1.3 UJPT Setup/Teardown

A UJPT hierarchy is created, extended, and deleted by using the create and delete interfaces for User, Job, Process, and Thread objects.

6.1.3.1 UJPT Setup

The creation of the first Job object for a MICA user is the event that triggers the creation of a UJPT hierarchy. Once established, a UJPT hierarchy is extended by creating additional Job, Process, and Thread objects. Figure 6–1 illustrates a complex UJPT hierarchy.

6.1.3.2 UJPT Teardown

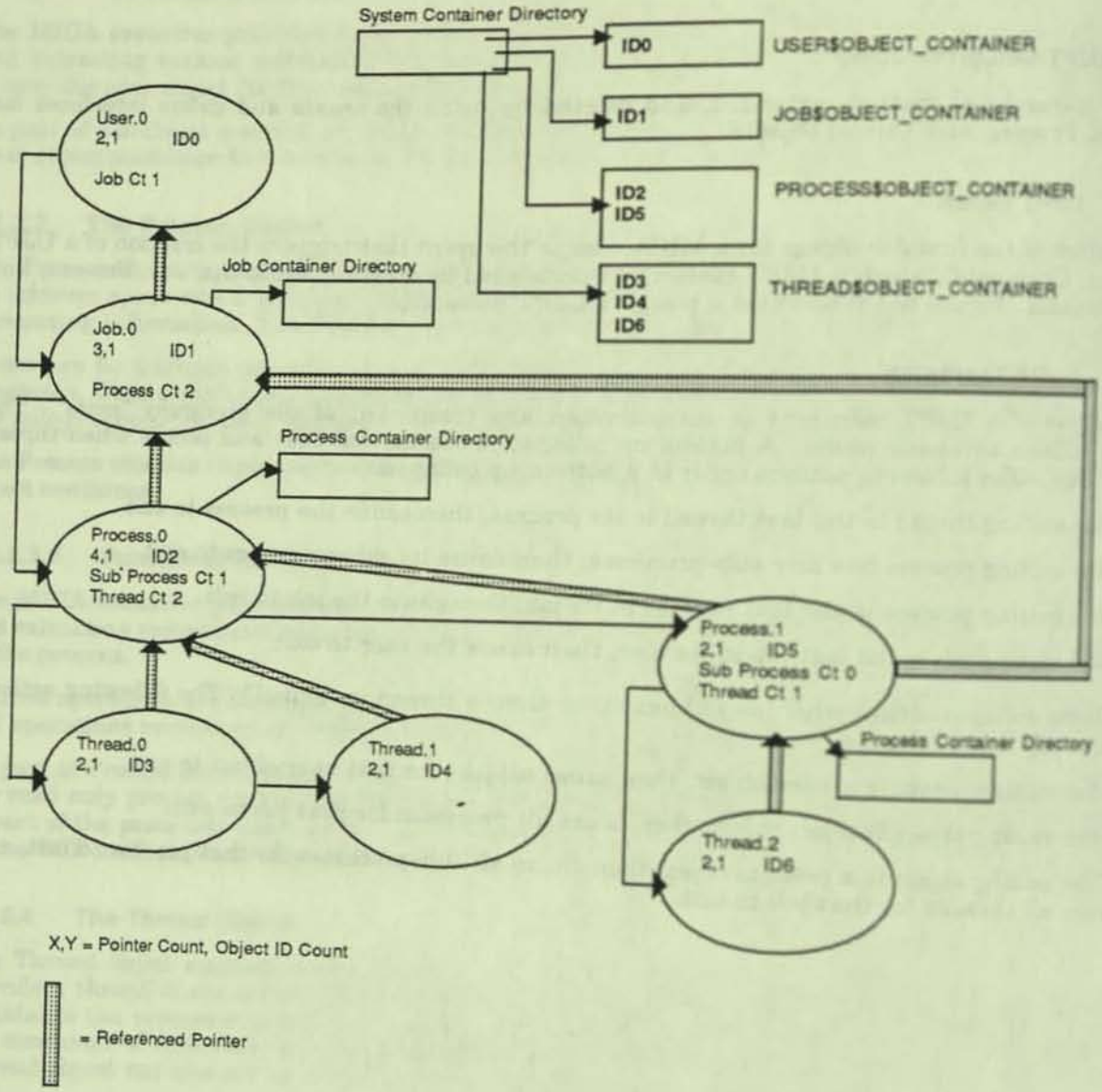
The collapse of a UJPT hierarchy is started when any component of the hierarchy "exits". The collapse follows two basic paths. A bottom up collapse is the normal case and occurs when thread objects "exit". The following actions occur in a bottom up collapse.

- If the exiting thread is the last thread in its process, then cause the process to exit.
- If the exiting process has any sub–processes, then cause its sub–processes to exit.
- If the exiting process is the last process in its job, then cause the job to exit.
- If the exiting job is the last job in its user, then cause the user to exit.

A top down collapse occurs when any object other then a thread is "exited". The following actions occur in a top down collapse.

- If the exiting object is a user object, then cause all jobs for that user object to exit.
- If the exiting object is a job object, then cause all processes for that job to exit.
- If the exiting object is a process object then cause all sub–processes for that process to exit, and cause all threads for that job to exit.

Figure 6–1: Complex UJPT Tree



ZS-2434747

CHAPTER 7

MEMORY MANAGEMENT

7.1 Overview

7.1.1 Requirements

The *memory management subsystem* provides a combination of hardware and software functions to accomplish the mapping of *physical address space* into the virtual *address space* of a process. The *physical address* is used by hardware to identify a page in physical memory. The memory management subsystem has six principal requirements:

- A number of processes may occupy main memory simultaneously, all freely using their own unique address spaces, while only accessing their own data and code.
- Only a portion of the total address space for a process needs to be resident at any one time.
- The data and code belonging to a process are scattered throughout physical memory and need not be contiguous.
- Processes can automatically share code and data.
- Processes are protected from themselves and from other processes.
- Support for the I/O system. This includes mapping of I/O space, and locking pages in memory for I/O.

7.1.2 Functional Description

The Mica memory management is designed to support a large user virtual address space (2 gigabytes per address space) ,and large working sets (4 gigabytes per address space) . Figure 7–1 illustrates the layout of the virtual address space associated with a process.

7.1.2.1 Environment of Memory Management

The memory management subsystem executes in kernel mode. Through the use of mutexes, multiple processors may be executing within the memory management subsystem simultaneously. During handling of the translation not valid fault, ASTs are disabled. This prevents an AST from interrupting the translation not valid fault processing, causing a recursive entry into the translation not valid code.

The memory management subsystem consists of the following features.

- Fault handlers for *access violation* – Checks to see if the offending page is a guard page for a user's stack. If so, the guard page is unprotected and a condition indicating the stack guard page was accessed is raised. Otherwise, an access violation condition is raised.
- *Fault on read*—Raises an access violation condition.
- *Fault on write*—Implements copy-on-modify semantics, and helps to track the modified state of a page.

- *Fault on execute*—Checks to see if the page is a kernel entry page for system service dispatching. If so, it saves appropriate registers, and calls the system service. Otherwise, it raises an access violation condition indicating that the user attempted to execute nonexecutable data.
- *Translation not valid*—Implements the pager. The page table entry for the faulting page is examined to determine how to make the page valid. The faulting page can come from a mapped file, a paging file, a page of zeroes, or a page that is already in memory. In the latter case, the page that is already in memory is either in a transition state, or shared with another address space that already has the page valid. This is also refered to as a *page fault*.

In addition, there are a number of system routines that contribute to memory management. These include:

- System services—Affect an address space
- Executive services—Manage and allocate pages from paged and nonpaged pools; also probe, lock, and unlock I/O buffers from memory
- Balance set manager—Ensures ample free pages
- Modified page writer—Writes modified pages

7.1.3 Memory Management Data Structures

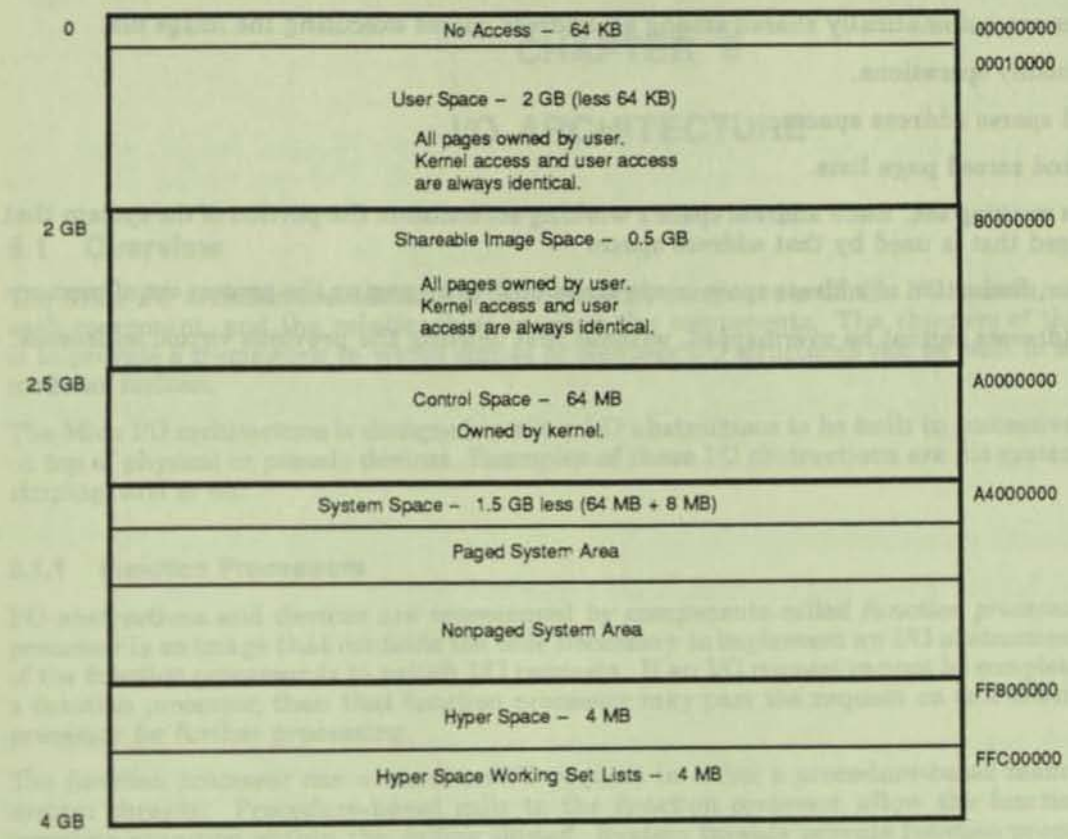
The following system data structures are used by the memory management subsystem:

- *Page frame number (PFN) database* —Tracks physical pages and their states. Each physical page is in one of five states:
  1. Active and valid—A page in this state is mapped in some address space's working set.
  2. Free—Available for immediate reuse.
  3. Zeroed—Available for immediate reuse.
  4. Standby—A page in this state is marked as in transition in a (prototype) page table entry (PTE), and may be reactivated as the result of a page fault for the transition page. This page can be reused, but the page table entry must change from a transition state to an invalid state.
  5. Modified—A page in this state is marked as in transition in a (prototype) PTE, and may be reactivated as the result of a page fault for the transition page. Before the page can be reused its contents must be written to disk. Once its contents are written to disk, the page enters into the standby state.
- *Working set list*—Manages the physical pages owned by an address space. Each address space is guaranteed a certain number of physical pages. When a page fault occurs which would cause that number of pages to be exceeded, a physical page is removed from that address space by making the page table entry invalid and decrementing a usage count for the page. The working set list is a list of virtual page numbers which are currently valid.
- *Page table pages* —Manage the complete address space. Each address space contains one segment 1 page table page which is one page in size and contains 512 page table entries. There are multiple, up to 512, segment 2 page table pages which are one page in size and contain 1024 page table entries.

  Each page table entry is a quadword and indicates whether a page exists at the corresponding address, and whether the page is valid or not. If a page is not valid, its page table entry indicates how to make the page valid.

- *Mapping objects*, *section objects* and *segment objects*—Track mapped files.

Figure 7–1: Virtual Address Space Layout



Notes on Figure 7–1:

- *User space*—Maps user code and data. Includes 64 KB that is set no access to catch programming errors.
- *Shareable image space*—Maps system-wide installed shareable images.
- *Control space*—Maps kernel-mode stacks and other thread-related structures.
- *System space*: Paged area—Maps pages that can be paged to disk. This area includes code, data, and pool.
- System space: Nonpaged area—Maps pages that must be memory resident. This area includes code, data, and pool.
- *Hyper space*—Maps address space page tables and data structures.

7.1.4 Differences from the VAX/VMS Memory Management Subsystem

Mica memory management supports several enhancements over VMS memory management, including the following:

- An address space's page table pages are only valid within that address space.
- Image files are automatically shared among all address spaces executing the image file.
- Copy-on-modify operations.
- Large and sparse address spaces.
- Standby and zeroed page lists.
- No system working set. Each address space's working set contains the portion of the system that can be paged that is used by that address space.
- No swapper. Reduction of address space use is accomplished by paging the process out of memory.
- Virtual addresses cannot be overmapped, without first deleting the previous virtual addresses.

CHAPTER 8

I/O ARCHITECTURE

8.1 Overview

The Mica I/O architecture defines the fundamental components of the I/O system, the interface of each component, and the relationships between the components. The objective of this architecture is to provide a framework in which simple or complex I/O structures can be built in an efficient and modular fashion.

The Mica I/O architecture is designed to allow I/O abstractions to be built in successive virtual layers on top of physical or pseudo devices. Examples of these I/O abstractions are file systems, shadowing, striping, and so on.

8.1.1 Function Processors

I/O abstractions and devices are represented by components called *function processors*. A function processor is an image that contains the code necessary to implement an I/O abstraction . The purpose of the function processor is to satisfy I/O requests. If an I/O request cannot be completely satisfied by a function processor, then that function processor may pass the request on to a lower-level function processor for further processing.

The function processor can execute an I/O request in either a procedure-based manner or by using system threads. Procedure-based calls to the function processor allow the function processor to complete execution within the calling thread. System threads provide function processors with the mechanism to do extended processing, including I/O waiting, after returning control to the user thread. System threads belong to the function processors that queue requests to them.

8.1.2 Objects Used by the I/O System

The Mica I/O architecture defines three I/O objects:

- Function Processor Unit (FPU) object
- Channel object
- Function Processor Descriptor (FPD) object

The functions of these objects are described in the following sections. The I/O architecture defines two significant data structures that are not owned by any particular object. These two structures are the *I/O request packet* (IRP) and the *I/O status block* (IOSB).

The IRP maintains the user's I/O request, as well as some bookkeeping information that is used by various components and objects in the I/O system. The IRP is allocated when an I/O request is made and is deallocated when the request is completed.

The I/O status block contains the final status of the I/O request and other data (such as byte transfer count) that is written to it when the I/O request completes.

8.1.2.1 FPU Object

A function processor accepts requests on one or more *function processor units* (FPUs). An FPU represents a particular resource to higher levels of software. All requests to a resource are directed to its respective FPU, which then specifies the appropriate function processor to process the request. Examples of these FPUs are the ODS II unit, shadow unit, striping unit, device unit, MSCP unit, and so on.

8.1.2.2 Channel Object

A *channel* object describes a logical I/O path to an FPU on which I/O requests can be issued. The channel object receiving the initial user request maintains a listhead of all outstanding IRPs. This listhead is only used in the event that all outstanding requests on this channel need to be canceled.

Channel objects are only associated with FPU objects and thread objects.

8.1.2.3 FPD Object

The *function processor descriptor* (FPD) object maintains the addresses of each global procedure in the function processor. The I/O architecture has a defined set of procedures that are common to all function processors. When the function processor is needed to process an I/O request, the address of the appropriate function processor procedure is looked up via the FPD object.

8.1.3 I/O Request Synchronization

The Mica I/O architecture supports two types of I/O requests:

- Synchronous
- Asynchronous

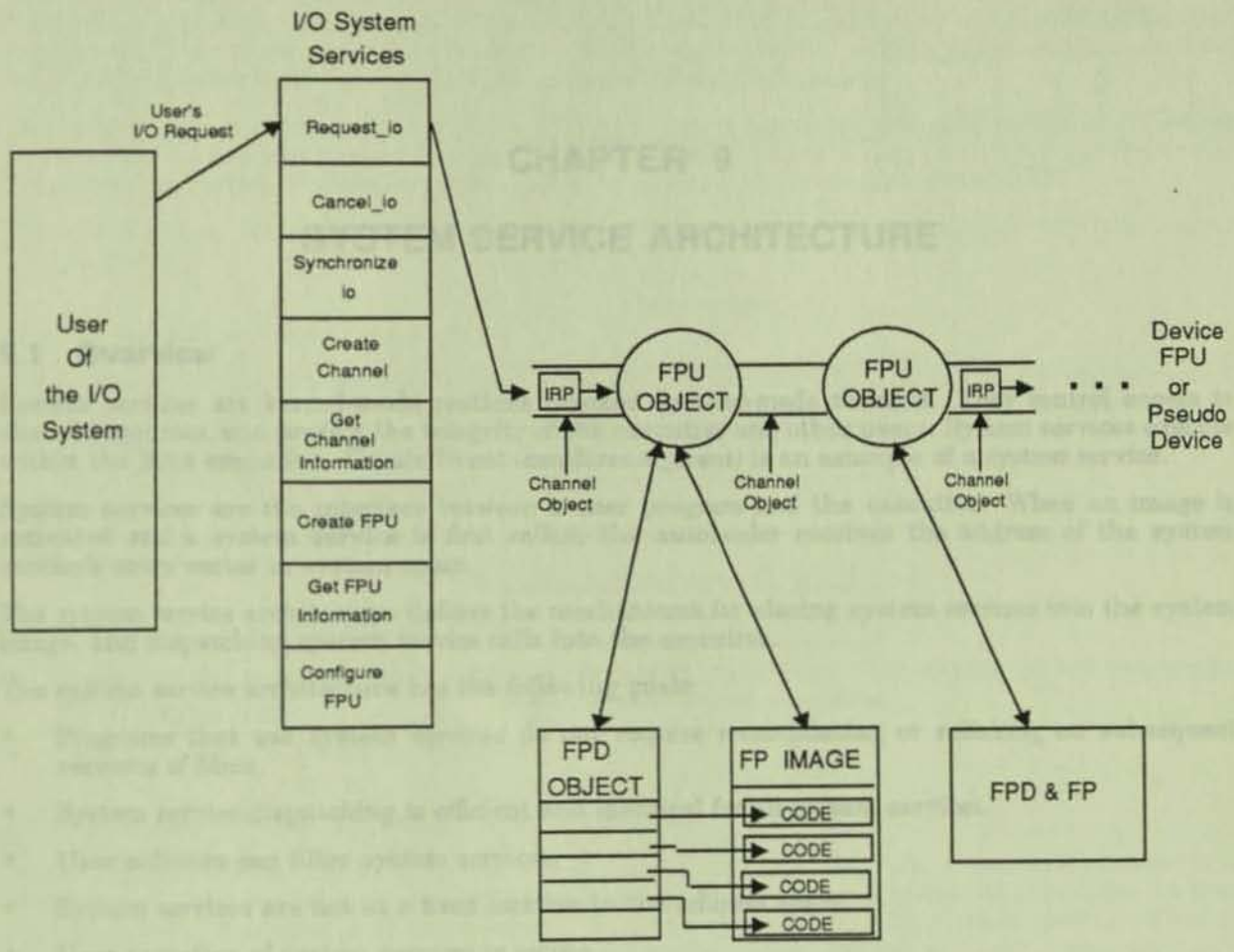
If a synchronous request is specified, the issuing thread is blocked until the request completes. If the request is asynchronous, then the issuing thread is not blocked, but continues to execute. The program issuing an asynchronous I/O request has the choice of specifying an AST procedure, an event object, or both to synchronize its execution with the completion of the request. When an asynchronous I/O request completes, the specified event object is signaled and/or the specified AST is queued.

8.1.4 I/O Service Routines

The I/O Architecture specifies a set of well-defined interfaces to the I/O system for the purpose of initiating, canceling, and synchronizing I/O requests; as well as for creating, manipulating, and deleting of objects. Some of these interfaces to the I/O system are available via system service routines. Other interfaces are designated as internal, and are only available to components of the I/O system, such as function processor and system threads.

The diagram in Figure 8–1 shows a typical configuration of the I/O system. Before the I/O system can be used, an FPD object must be created for each function processor and an FPU object must be created for each available resource. After the system has been set up, the user can then create a channel object to an FPU, and issue an I/O request via the Request_IO system service routine. The user is notified when the request has been satisfied.

Figure 8–1: Overview of Mica's I/O Architecture



8.1.5 I/O Security

I/O requests are subject to access permission checks by Mica security and I/O system support routines. Mica security provides mechanisms for granting and denying access to channel and FPU objects. Additional security checking is done by the I/O system support routines to determine if the function code specified for the I/O request can be issued over the channel.

See Chapter 10, Security and Privileges and Chapter 8, I/O Architecture for more information.

CHAPTER 9

SYSTEM SERVICE ARCHITECTURE

9.1 Overview

System services are kernel-mode routines invoked by user-mode threads. They control access to shared resources, and protect the integrity of the executive and other users. System services execute within the Mica executive. Create Event (*exec$create_event*) is an example of a system service.

System services are the interface between a user program and the executive. When an image is activated and a system service is first called, the autoloader resolves the address of the system service's entry vector in system space.

The system service architecture defines the mechanisms for placing system services into the system image, and dispatching system service calls into the executive.

The system service architecture has the following goals:

- Programs that use system services do not require recompilation or relinking on subsequent versions of Mica.
- System service dispatching is efficient and identical for all system services.
- User software can filter system services.
- System services are not at a fixed location in the address space.
- User execution of system services is secure.

9.1.1 Functional Description

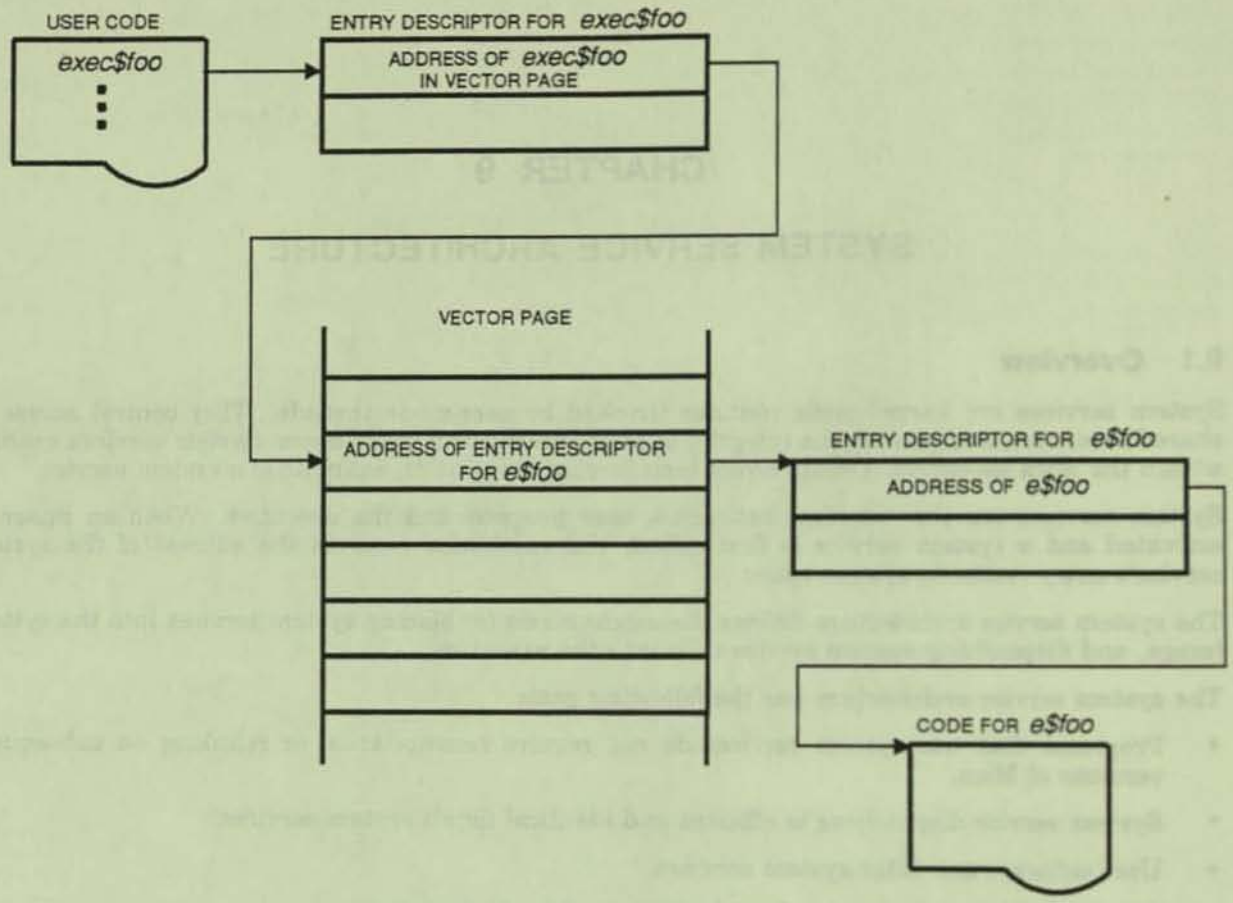
When an image is built that calls system services, Mica resolves the references to the system services from a shareable image. The system service shareable image contains the entry descriptors for all system services. The autoload mechanism, discussed in Chapter 31, Image Activation and Chapter 30, Linker, is used to resolve the references to system services to their proper addresses within the Mica executive.

When the autoload routine loads the system service shareable image into the process's address space, it performs the shareable image fixups. The fixup for the entry descriptors involves adding the base address of the system service vector page to the entry descriptor found in the shareable image.

The system service vector page resides in the nonpaged portion of the system address space, protected as user read, kernel read, fault on execute. When a user calls a system service, a JSR instruction executes, using the address located in the entry descriptor as a target. In this case, the address is within the system service vector page. Figure 9–1 depicts the flow of control involved in this process.

Since the system service vector page is valid and has fault on execute enabled, a fault on execute (FOE) fault is generated. The fault vectors through the SCB to the system service dispatcher.

Figure 9–1: Dispatching System Services



9.1.1.1 System Service Dispatcher

The system service dispatcher analyzes the page table entry (PTE) for the faulting address, and if the page is not a kernel entry page for the system service, an access violation is reported. Otherwise, the system service is dispatched.

The dispatching consists of:

- Saving the appropriate registers so the system service can be repeated.
- Loading R3 with the address of the thread control block for the current thread.
- Loading the first longword of the thread control block with the previous mode argument from the PS.
- The address which was the target of the JSR instruction contains the address of the entry descriptor for the system service. The linkage mechanism is set up and the system service is called at its entry point.

9.1.1.2 The System Service

The code for each system service is responsible for ensuring that all arguments are probed and captured as necessary. Sil V2.0 and Pillar will have some support for probing and capturing, but that support will not handle item list elements or record elements. All writers of system services must ensure that all arguments are probed and captured as appropriate.

The system service must declare a handler to catch access violations and other conditions that can be raised by the executive or kernel. Any conditions not handled are caught by the prebuilt handler for the system service dispatcher. The handler for system services causes a bug check.

When the system service completes, it returns to the system service dispatcher with the system service status.

9.1.1.3 System Service Completion

Upon return, the system service dispatcher checks the status values. If the status values are not either repeat service or resume service, the service is complete and will return to the caller. The return is accomplished by:

- Restoring R3 and the frame pointer to their original contents
- Setting the linkage registers for the return
- Clearing any volatile registers which could contain sensitive information
- Popping the stack back to the PC/PS pair
- Modifying the original PC to return to the caller rather than the address within the system service vector page.
- Issuing an REI to return to the caller

9.1.1.4 Repeatable and Resumable System Services

When a system service issues a wait, it has the option to accept delivery of user-mode ASTs. This is accomplished by declaring to the kernel wait call that the service is willing to accept the "deliver user-mode AST" condition and has handlers set up to clean up from the system service. Services that accept delivery of user-mode ASTs fall into two categories. *Repeatable services* are completely reexecuted after the delivery of the ASTs. In contrast, *resumable services* cause another system service that performs a continuation of the original service using the original arguments.

The following steps occur if a "deliver user-mode AST" condition is raised:

- The kernel unwaits the waiting thread with the status "deliver user-mode AST".
- The "deliver user-mode AST" condition is raised.
- The system service condition handlers clean up the service by deallocating any resources, decrementing counters, etc. This is necessary because the REI transfers control to user-mode code and the system service does not have to be repeated.
- The system service dispatcher notices that the returned status is either "repeat" or "resume" and restores the argument registers, etc.
- If the status is "resume system service," the system service dispatcher increments the PC stored in the kernel stack by four, and stores it back in the kernel stack, so that an REI will cause the next entry vector to fault.
- The system service dispatcher pops the stack back to the PC/PS pair.

  At this point the registers contain the same values they contained when the system service dispatcher was invoked.
- The system service dispatcher executes an REI instruction.

- When user mode is restored by the REI instruction, the user-mode AST is delivered. Before the AST is executed, the current context is saved, and after the AST is completed, the context is restored. Part of the context that is saved and restored is the current PC, which is the address of the system service's entry vector.
- When the user-mode AST is completed, execution is attempted at the entry vector, and the resulting FOE fault dispatches to the desired system service which repeats or resumes the execution of the system service.

9.1.2 Changes to the Existing Chapter

The following changes will be made to the existing chapter:

- Remove system service entry page. System services will be autoloaded.
- Remove previous mode argument. The previous mode argument is the first longword of the TCB.
- Remove the fixed system service vector page. This page is built at system initialization and the autoloader knows how to find it.

CHAPTER 10

SECURITY AND PRIVILEGES

10.1 Overview

This paper is the overview of the Security and Privileges Chapter of the Mica Working Design Document. The overview describes the security model for the base Mica operating system. The security requirements of the compute server are not discussed specifically as they are believed to be a subset of the security model presented in this overview. The security requirements of the database server are not discussed at this time because they are not known.

The security model is described in terms of authentication, access control, and security audits. It does not assume the existence of workgroups. It treats a Mica system as an independent security domain with its own authorization database. A user must be entered in the authorization database in order to gain access to the system. Any objects that are located on Mica and are accessed on behalf of the user are accessed with the user's Mica access rights.

10.1.1 Authentication

When a user attempts to gain access to the system, the system must verify the identity of the user. The act of verifying the identity of a user is called *authentication*. Requests to gain access to the system originate from many sources, including a request to connect to a server from a local or remote node, or a request to create a process that has a different username. On Mica, the software that manages requests to gain access to the system must authenticate the request before allowing access. If the authentication succeeds, the software can allow the user access to the system. If the authentication fails, the software cannot allow the user access to the system.

The exact method used to authenticate a user is still under design. One method is to have each server perform authentication. Another method is to have one centralized protected sub-system perform the authentication. Whatever method is selected, the checks performed would include verifying that the user is a valid user of the system, and checking the day of the week and time of day that the user is allowed access. Other checks can be added if they are needed. The information for user authentication is kept in a system authorization file.

On Mica, access to the system from remote nodes is via DECnet-like proxy access. If the remote user's nodename-username pair is not registered in the system authorization database, the user is refused access. If the user's nodename-username pair is registered, the user gains access.

10.1.2 Access Control

After a user has gained access to the system, the user can access resources that are on the system. On Mica, a resource can have no protection, so that all users can access it, or the resource can have protection so that only certain users can access it. The system controls access to resources by checking the user's access rights against the resource's access control information to verify that the user has access to the resource.

On Mica, any resource that needs to be protected must be an object as described by the Object Architecture. The access to a resource is checked indirectly by the procedure E$REFERENCE_OBJECT_BY_ID. Software calls E$REFERENCE_OBJECT_BY_ID and passes it the object ID of an object to obtain the address of the object (reference the object). One of the things that E$REFERENCE_OBJECT_BY_ID does is call the procedure E$VERIFY_ACCESS to verfiy that the caller has access to the object. Therefore, whenever software references an object, the access to the object is checked. This is the only place in Mica that E$VERIFY_ACCESS is called.

10.1.2.1 User Access Rights

Each user that is allowed to use a Mica system is given a set of access rights. Access rights represent the user's claims to resources on the system. On Mica, the access rights are kept in each thread that is owned by a user.

\ Are access rights a per-process or a per-thread attribute? Can threads within the same process have different access rights? If a server handles requests for only one client, the access rights should be kept in each process; then all threads within one process would have the same access rights. On the other hand, if a server handles requests for multiple clients, each thread could assume the access rights of a client; in this case, each thread has different access rights and they should be kept in each thread. \

The access rights of a user is made up of two parts: a mode and an identifier list. The mode is the processor mode that the thread is executing in. On Mica, the mode is either user or kernel. The identifier list is a list of 32-bit values, called identifiers, that represent who the user is and what groups the user is a member of. Each time a user gains access to the system, the user is assigned the same identifiers. Each identifier also has an alphanumeric name at the human interface level.

Each user is assigned a unique identifier, called the user identifier, that identifies the user to the system. Note that this is different from VMS where a user is identified by a UIC. This identifier is always included in audit messages.

Privileges are not included in the access rights of a user because there are no privileges on Mica. Privileged access to objects is implemented using identifier lists and access control on objects. For example, privileged executive code could be implemented as an object. The user would have to have the appropriate identifiers to access the object and execute the code.

10.1.2.2 Object Access Control Information

As stated above, resources on Mica that need protection are required to be objects. Each object has access control information that describes the access rights needed by a user to gain access to a resource. On Mica, the object header of each object contains the access control information.

The access control information of an object is made up of three parts: a mode, an owner identifier, and an access control list (ACL). The mode is the processor mode that the object was created in. The owner identifier is the user identifier of the user who created the object. The ACL is a list of one or more access control entries (ACE). Each ACE contains a list of one or more identifiers and the access allowed to the object. If a user has the identifiers listed in the ACE, and the access requested is a subset of the access that is allowed by the ACE, the user is allowed the requested access.

10.1.2.3 Access Control Algorithm

When a user tries to access an object, the system examines the user's access rights and the object's access control information to determine if the user has access to the object. The following are the steps taken by the system to determine access: (Note that "desired access mode" is the processor mode of the user at the time the user requested access to the object; it is not the current mode of the thread because when the system checks access to an object, the current mode of the thread is always kernel.)

1. If the desired access mode is kernel, access is allowed.
2. If the desired access mode is user and the object does not have an ACL, the access is determined from the mode of the object:
   - If the mode is user, access is allowed.
   - If the mode is kernel, access is denied.
3. If the desired access mode is user, the object has an ACL, the user is the owner of the object, and the access desired is CONTROL, the ACL is ignored and access is allowed.
4. If the desired access mode is user and the object has an ACL, the access is determined from the ACL. The system examines each ACE in the ACL until either an ACE is found whose identifiers are all listed in the user's identifier list, or the end of the ACL is reached:
   - If an ACE is found, the access requested by the user is checked against the access allowed by the ACE. If the requested access is a subset of the access allowed by the ACE, the user is allowed access. If the requested access is *not* a subset, the user is denied access.
   - If an ACE is not found, the user is denied access.

10.1.3 Security Audits

Security audits allow certain system events to be auditted. These events include the login and logout of users, the mounting and dismounting of devices, and the successful or unsuccessful access to objects. Currently on Mica, the only system event that will be auditted is access to objects.

The auditting of access to objects is implemented by audit entries that are located in an object's audit list. On Mica, the object header of each object points to the audit list. Each audit entry contains the type of access to audit and the names of one or more audit sinks. When access to an object is allowed or denied, the system checks all the audit entries in the audit list to determine if the access request should be auditted. Mica will use the message function processor to collect and disperse audit messages to the appropriate audit sink.

The software that handles the messages in an audit sink determines the characteristics of the sink. For example, the software that handles a log sink would write the messages to a disk file. The software that handles an alarm sink would write the messages to a security terminal or console.

Audit messages are generated by E$VERIFY_ACCESS. No other procedures are allowed to generate audit messages.

10.1.4 Issues

- The Mica operating system will *not* have the capability and functionality to apply for a C2 class security rating as defined by the Department of Defense. Why? Because we do not have a secure communication channel between a client system and a Mica system. Plus, there is no way to verify a client's "connect-to-server" request to make sure that the request actually came from the client.
- If the assumption is made that a client's "connect-to-server" request actually came from that client, how is the user authentication done? Does each server perform user authentication or is there a centralized authentication service?

- The access control requirements of the database server are not known.
- Is there a one-to-one or a many-to-one relationship between a client and server? If there is a many-to-one relationship, the server could have minimal access rights so that the threads that service client requests assume the access rights of the client when accessing objects. Or, the server could have maximum access rights (enough to access all objects) so that threads servicing client requests check to ensure the client has access to the object. If the client has access, the thread accesses the object using the server's rights.
- This specification may not play well with workgroups.
- Does an ODS 2+ disk on a Mica system exhibit the same access control behavior as an ODS 2+ disk on a VMS system?
- How does a user on a client system create, modify, or delete ACLs for objects on a Mica system?

CHAPTER 11

CONDITION, EXIT, AND AST HANDLING

11.1 Overview

This chapter describes four facilities: condition handling, unwinding, support for exit handlers, and support for user-mode asynchronous system trap (AST) handlers. The chapter specifies goals, interfaces, and algorithms for the areas. These facilities are all related and are designed to work together.

There are currently no outstanding issues for this chapter and the only planned modifications are those resulting from the last group-wide review, with the addition of a section on user-mode AST handlers—previously undocumented for Mica.

Each facility is described separately.

11.1.1 Condition Handling

A *condition* results from an error encountered during thread execution. It may be due to a hardware or software failure. Examples of such hardware errors are arithmetic traps, access violations, and so on. Examples of such software errors are range checking, argument checking, and so on. The Mica *condition handling facility* provides the capability for programs to process such conditions in a controlled fashion.

A *condition handler* is a procedure written as a part of a program or supplied by a run-time facility to handle conditions if they occur during the execution of that program. Should a condition occur during program execution, Mica must be able to find a thread's condition handlers. The condition handling facility provides the mechanism by which handlers are found and established (either at runtime or compile time).

When a condition occurs, it is said to be raised in the thread which caused it. Raising a condition interrupts the normal control flow in a thread, saves its context, and causes a search to be made for a condition handler established by the thread. If a handler is found, it is called as a procedure, with arguments describing the nature of the condition (the *condition record*) and the environment in which it occurred (the *mechanism record*).

A condition handler may choose to handle the condition (that is, perform some actions relating to the condition) or may choose to reraise the condition (normally done for conditions which that handler is not written to handle). In the second case, the search for handlers continues and the next handler found is called. This process continues until some handler either indicates that the thread should continue (either from the location of the condition or using the unwind facility from a different location) or causes the thread to exit, or until no more established handlers can be found. In this last case, the *system catchall handler* is called.

There are three types of condition handlers:

- Vectored handlers
- Invocation descriptor-based handlers

- The system catchall handler

There may be many vectored handlers or invocation descriptor-based handlers, none of which are supplied by Mica. There is only one system catchall handler, which is always provided by Mica.

Vectored handlers may only be established at runtime, by using a system service. There are two types of vectored handlers: primary and last chance. Primary vectored handlers are the first searched for when a condition is raised. The list of primary handlers is called in FIFO order with respect to when they were established. If all have been called and reraised, the invocation descriptor-based handlers are then called for currently active procedures, from the most recently active to the oldest. Finally, if all these reraise the condition, the last chance vectored handlers are called in LIFO order with respect to when they were established. Should all these reraise as well, then the system catchall handler is called, which produces an error message and causes the thread to exit.

Invocation descriptor-based handlers are established at compile time. They are located from a procedure's invocation descriptor. These handlers are used to implement a particular language's condition handling semantics. For the Pillar language, they are used to implement structured condition handling.

Mica condition handling is designed to allow the processing of nested conditions and also to handle boundary problems with stack limitations. It also provides the following additional features:

- Invocation descriptor-based handlers may be called multiple times when multiple conditions are active. This behavior may be enabled per handler. (Note that this is required for PL/1 support.)
- Environment information relating to a condition contains the set of scratch registers used in a PRISM procedure call, together with the stack pointer (SP) and frame pointer (FP) at the time of the condition (that is, registers R1 through R31, inclusive).
- Condition information is complete, including information relating to message files and argument typing.
- A separate stack is available for the execution of vectored handlers. This improves the capabilities of the Mica debugger.

11.1.2 Unwinding

The Mica *unwind facility* centrally provides the capability to perform nonlocal GOTOs within a thread. It is implemented as a user-mode procedure, mapped in system space, and reached via a procedure variable in the process control region (PCR).

A call to the Unwind service specifies a target procedure and point in that procedure from which to continue thread execution. The target procedure must be an ancestor of the calling procedure. A target procedure invocation is specified either by its stack frame pointer (procedure invocations without stack frames may not be unwound to), or as the caller of the establisher of the last active condition handler. A condition record may be specified along with this target to give information relating to why the unwind operation is taking place.

Prior to returning execution to the target procedure invocation, the unwind facility searches for and calls any invocation descriptor-based condition handlers established for any procedure invocations found between the calling procedure and the target procedure invocation. These are called with the condition record and a mechanism record (constructed by the Unwind service) relating to where the unwind request was made. This allows procedure invocations that are being discarded a chance to clean up; for example, to deallocate any virtual storage they have allocated.

Once this phase has completed, the target invocation's register context is restored and the execution is continued from the specific point. Note that in the case of an unwind after a condition handler has been active, R8 and R9 are restored from the mechanism record, allowing a return status to be set.

Since processes in Mica are multithreaded, it is necessary for each thread to clean up its use of the common address space. The unwind algorithm is designed to help this take place: instead of exiting a thread by using the Exit system service, a call to Unwind is made, specifying the beginning of the call hierarchy as the target. Unwind then calls all established invocation descriptor-based handlers, causing them all to clean up their own environments. When the beginning of the call hierarchy is reached, Unwind calls the thread Exit system service, with the input condition record argument as status.

Thus, in Mica, user-mode thread exit is accomplished using the unwind facility, not by using the thread Exit system service directly.

11.1.3 Exit Handling

The Mica *exit handling facility* allows threads and processes to perform overall clean-up actions on their environment or deallocation of system resources. *Exit handlers* are procedures established by a thread during execution and called in user mode after a thread has called the thread (or process) Exit system service. There is no way in Mica a thread or process can exit without attempting to call exit handlers.

There are two types of exit handlers: thread and process exit handlers. Thread exit handlers are called when a thread exits. Process exit handlers are called when the last thread in a process has finished executing the last of its thread exit handlers.

Exit handlers are established using a system service and kept as a list in either the PCR or the thread control region (TCR). This helps ensure that the exit handler list cannot be accidentally corrupted. The lists are called in LIFO order. Each list entry has a procedure variable and a nontyped 64-bit parameter, which may be used to pass information to the handler when it is activated during thread exit. Entries may only be removed by using the Establish Exit Handler system service with the appropriate arguments.

Exit handlers are called with the 64-bit, user-specified argument kept in the list entry and a condition record. This is the condition record that was used in the call to the thread Exit system service which activated the exit handler. An exit handler completes it processing by calling the thread Exit system service. Thus, each call to the Exit system service removes a handler from the list and calls it, until the list is empty, in which case the rest of thread rundown continues.

If, during the execution of an exit handler, a forced exit request is made for that thread, then the current exit handler is terminated, and the next one on the list is called. All handlers are allowed to run until they exhaust CPU quota. They may not establish new exit handlers. Should an exit handler exceed the thread's CPU quota, it is terminated. The thread's CPU quota is then incremented by a fixed amount and the next handler found and called.

Note that a forced exit request for a thread which is not executing exit handlers causes an exit unwind operation to occur prior to calling any exit handlers.

11.1.4 User-Mode AST Handling

The Mica user-mode *AST handling facility* provides a mechanism for delivering asynchronous event notification in user mode to threads. Many Mica system services have the capability of executing in parallel with a thread's execution and/or causing subsequent asynchronous event notification to the thread. Thus, a thread may issue a system service, the service may return with a pending status, and the thread may continue executing. When the service later completes the requested action or an event associated with that service occurs, if the thread established an AST handler in the service call, a user-mode AST is queued to the thread. The AST is delivered as soon as the thread is next eligible to run, unless an AST has already been delivered and is being processed by the thread, or if the thread has disabled the delivery of user-mode ASTs.

An *AST handler* is a procedure that is intended to receive such notification. These procedures are part of the program and are associated with a particular event or system service completion notification required by the thread during its execution. An AST cannot occur unless the thread has established an AST handler for it.

To establish an AST handler, a thread uses a procedure variable for the handler in the system service call that can cause the desired AST. Along with this procedure variable, the thread may specify a 64-bit quadword untyped parameter. When the procedure is subsequently called to process the AST, this parameter, together with an AST-specific, 64-bit quadword untyped parameter, is used as an input argument. These arguments are used to identify the AST and to pass information to the thread concerning the AST.

Once a user-mode AST has been delivered, no other user-mode ASTs can be delivered until it has finished being processed. Subsequent ASTs are blocked by hardware until the thread explicitly leaves AST state, thereby removing the block. The AST is delivered to system-supplied code in user mode. This procedure sets up a stack frame and then, in turn, calls the specified AST procedure with the AST parameters. When the AST procedure returns, the system procedure uses a system service to remove the AST In Progress flag for the thread and dismisses the AST state, allowing the delivery of further user-mode ASTs (if any are pending). The stack is then cleaned and an REI instruction used to continue the thread's previous execution.

Note that the "false" stack frame is important: it is used to provide continuity when attempting an unwind through an AST event. The system procedure has an invocation descriptor-based condition handler established specifically to deal with this possibility.

At any time, a thread may disable or enable the delivery of user-mode ASTs. This is accomplished using the SWASTEN instruction and does not involve any system services.

11.1.5 Dependencies

This chapter depends on the:

1. PRISM SRM—specifically, hardware exceptions
2. PRISM Calling Standard
3. Mica process architecture design
4. Mica kernel design

Note that the Mica debugger design depends on this chapter.

CHAPTER 12

BOOTING

12.1 Overview

This chapter discusses the bootstraping process for PRISM processors, how the Mica system uses this process to bootstrap itself, and finally, the provisions made that allow operating systems other than Mica to bootstrap themselves.

NOTE

It is a goal of the PRISM bootstrapping process to remain completely decoupled from the existence of and external service processor.

12.1.1 Bootstrap Structure

The purpose of the bootstrap process is to define a process capable of handing over a cold machine to system software. On PRISM processors, this bootstrap process occurs in three phases.

1. Hardware Bootstrap
2. Primary Software Bootstrap
3. Secondary Software Bootstrap

The three phases of bootstrap are responsible for initializing the PRISM processors to a known and architecturally defined state, loading and passing control to an operating system independent primary bootstrap program, and finally loading and passing control to an operating system dependent secondary bootstrap program.

12.1.2 Hardware Bootstrap

The hardware bootstrap is defined by the PRISM System Reference Manual. The purpose of the hardware bootstrap is to:

- initialize each PRISM processor to an architecturally defined state
- initialize portions of system memory to an architecturally defined state
- load the PRISM primary software bootstrap program (PSB) into system memory, and pass control to it.

12.1.3 Primary Software Bootstrap

The primary software bootstrap is implemented as PSB. It is intended to be a relatively operating system independent piece of software. It will however contain some **ODS–II** specific file system primitives.

The **ODS–II** file operations exist to support Mica booting. Ultrix booting is accomplished through *logical block booting*; an **ODS–II** file system does not have to exist on the boot device to support Ultrix booting.

The primary software bootstrap is responsible for:

- Determining system type and performing system specific initialization.
- Creating an allocated physical memory descriptor which describes all of physical memory, allocated memory, and bad memory.
- Sizing and testing available memory.
- Initializing a System Control Block (SCB) for the bootstrap master processor.
- Determining the bootstrap device to be used by the secondary software bootstrap by searching the IO space, and examining values stored in the system–wide restart parameter block.
- Initializing the bootstrap device drivers, and creating a bootstrap device driver interface descriptor such that the bootstrap device drivers may be used by the secondary software bootstrap.

  The bootstrap device drivers export a standard read logical block interface. The drivers run at an elevated IPL. The drivers are modeled after the VAX/VMS bootstrap device drivers. They are not FPU based, or otherwise related to the IO device drivers present in the Mica system. The drivers provide a read logical block interface to possibly two boot devices.
- Creating a primary and an alternate path to the read–write and read–only system disks.

  This step is only performed if the underlying file system available through the primary interface is **ODS–II**. The purpose of the primary and alternate interface is to provide a "search list" capability for loading system files.
- Locate and load the secondary bootstrap program.
- Transfer control to the secondary bootstrap program.

12.1.4 Secondary Software Bootstrap

The secondary software bootstrap program is an operating system specific bootstrap. The secondary software bootstrap program has different responsibilities for different operating systems. There are currently two forms of secondary software bootstraps.

- Ultrix logical block bootstrap
- Mica secondary software bootstrap

12.1.4.1 Ultrix Secondary Bootstrap

In the case of Ultrix booting, the secondary bootstrap consists of a single 512 byte boot block that is loaded and transfered to. Once active, the program will load a "real" secondary software bootstrap which is intelligent enough to understand the Ultrix file system. The "real" secondary software bootstrap is responsible for loading, and transferring control to Ultrix.

NOTE

The restriction that forces the load of a single boot block is due to the PRISM SRM. The SRM defines a single location in the system–wide RPB to be used as a logical block number of a boot block. There are currently no provisions for storing a "number of blocks" parameter thus forcing an additional stage for Ultrix booting.

12.1.4.2 Mica Secondary Bootstrap

The secondary software bootstrap for the Mica system is implemented in *[sys$kernel]mica$sysboot.exe*. It is responsible for loading the portions of the Mica operating system that are required to initialize the system and load the modules required by the system.

12.1.5 Mica Bootstrap Summary

The following summarizes the flow taken during a bootstrap of the Mica Operating System.

- The hardware bootstrap occurs to initialize the PRISM processor to the state described in the PRISM SRM.
- PSB is loaded and invoked. PSB further initializes the system. This includes establishing a bootstrap device driver which supports a logical block read interface to the primary and alternate system disks.
- PSB uses the bootstrap device driver to load ("activate") the Mica SSB implemented in *[sys$kernel]mica$sysboot.exe*.
- The Mica SSB is invoked. SSB opens the file *[sys$kernel]mica$components.dat*. For each file name stored in this file, its image is loaded and "linked" to the other components loaded in with the initial system. After all files have been processed, SSB transfers control to *mica$system_initialize()*, the initial entry point of Mica.
- The *mica$system_initialize()* entry point is responsible for initializing both the core Mica system and the non–core Mica system. Once initialization is complete, the initial user–mode process is created. This process is implemented in *[sys$kernel]mica$startup.exe*.

CHAPTER 13

SYSTEM DUMP ANALYZER AND SYSTEM DEBUGGER

13.1 Overview

The System Dump Analyzer and the System Debugger are tools used to probe and debug a Mica system. The System Dump Analyzer is a utility that is used to help determine the cause of system failures. This utility can also be use to examine a running system. The System Debugger is a interactive debugging tool that is used to monitor the execution of the Mica operating system.

The primary group of users of both the tools are system programmers. The System Dump Analyzer could also be used by system managers to diagnose system problems including performance analysis.

13.1.1 System Dump Analyzer

The System Dump Analyzer (SDA) reads, formats, and displays the contents of the system dump file or a running system.

When a fatal error causes a system failure, the Mica operating system writes information concerning its status to a system dump file. This file contains a copy of the contents of memory and a copy of the hardware context at the time the failure was detected.

In addition to examining the contents of the system dump file, SDA can also examine the currently running system. This feature is not committed for FRS.

The System Management interface is used to activate the System Dump Analyzer.

13.1.1.1 Requirements & Goals

The requirements for the System Dump Analyzer are:

- A mechanism for customers to supply Mica failure information to the support organization
- A mechanism for Mica developers to analyze system failure

The System Dump Analyzer goals are to:

- Examine the data structures symbolically
- Allow symbols to be defined
- Have a command line interface compatible with the symbolic debugger being developed by the Software Development Technologies (SDT) group at Spitbrook
- Write system dump files to disk other than the system disk
- Write system dump files to tape in addition to disks.
- Display summary information about Mica and its sub-components
- Easily add new functions to the utility

13.1.1.2 Design Highlights

The code that writes the system dump file is activated by the system crash mechanism. When a system error has happened, the system crash mechanism is responsible for deciding whether the error is fatal and what actions need to be taken. The system crash mechanism always attempts to write errorlog information into a special preallocated errorlog crash file. Optionally the system crash mechanism may write a system dump file. A system dump is not written if

- The fault of the crash is know to be a hardware problem or
- The writing of dumps is disabled by TBD mechanism or
- A preallocated dump file does not exist

The system dump file writer is strictly a slave to the system crash code, and works at IPL 7. It cannot depend on the failed system to perform any functions. The system dump file writer uses the bootstrap device drivers. All the code which is used to write a dump file is checksumed and before a dump is written the checksum is checked.

Because of the large amount of memory that a Prism system can support, only part of the system memory is written to the system dump file. A subset algorithm specifies what part of the system memory to save. The contents of memory are divided into logical groupings,

Virtual memory database
Pooled memory
System space
Current process' memory
Server processes' memory
Memory associated with other processes

These groupings are prioritized as listed above, with essential parts always being written to the system dump file and non-essential parts being written as space allows in the system dump file.

A dump file for a system is preallocated and is overwritten. A system manager must take an action to save a dump file.

13.1.1.3 Issues

1. Is the subset memory dumps sufficient to analyze crashes?

   Not all systems failures can be isolated by examining dumps. Using the subset memory system dump files, there is a strong possibility that fewer failures can be isolated by examining dumps. However, the use of subset files may not greatly impact the usefulness of system dump mechanism.

   One alternative to subset dumps is after a crash, boot a "small" system which runs only SDA and can examine the contents of memory directly. However it is not clear is this alternative meets the requirements or goals of SDA.

2. Can a system memory get corrupted in such a way that it is impossible to identify the subset groups?

3. Possible primitive user interface at FRS

   Because of scheduling constraints, and since Mica development group members are the only supported system programmers at FRS, the user interface may be primitive at FRS.

13.1.2 System Debugger

The System Debugger (SD) is a debugging tool that is used to monitor the execution of the Mica operating system and user programs. The user of the SD can interactively examine memory, deposit values in memory locations, set breakpoints, perform single-step execution, define symbols, and evaluate arithmetic expressions.

Unlike the other debugging tools supplied by Mica, the system debugger works in the harsh environments in kernel mode at non-zero IPL. The SD is therefore very primitive. The debugger is self contained and does not use the Mica system which it is debugging for any services.

Although SD will be used for a short period of time to debug user mode programs, the preferred debugger for user mode programs is SDT's debugger.

13.1.2.1 Requirements & Goals

The requirements of the SD are that it:

- Runs at any IPL
- Runs in both user and kernel mode
- Does not use the Mica services
- Does not require special hardware

The SD goals are to:

- Allow symbols to be defined
- Do instruction decoding
- Have a command line interface compatible with the SDT debugger

13.1.2.2 Design Highlights

The implementation of the SD is straightforward. The SD performs all of its functions without the assistance of the Mica operating system.

The SD is loaded as part of the system, and is hooked into the system as an interrupt service routine (ISR) in the system control block (SCB).

The SD uses the console terminal as its user interface. The console terminal is the only I/O device used by the SD. The SD uses polling to perform the I/O, and does not employ interrupts.

The Mica System Debugger commands are less cryptic than VMS' XDELTA commands, and are compatible with the SDT debugger command line interface.

Currently there are no plans for SD to make use of the debugger symbol table included in an image.

Two side effects of the debugger implemetation are:

- Use of the SD requires a standalone system because the entire system is stalled when a breakpoint happens.
- Only the contents of physical memory may be examined. Virtual memory which is not in physical memory can not be examined or altered (including setting breakpoints).

13.1.2.3 Issues

1. The SDT debugger runs in Kernel Mode at IPL 0

   It is a goal to have the SDT debugger run in kernel mode at IPL 0. Therefore, the only environment in which the SD would have to be used is at kernel mode at non-zero IPL.

2. A command line interface like that implemented by the SDT parser may be too complicated.

   In the context of the SD, the parser needed to handle SDT commands may be over complicated. Therefore, the need for a parser may be in opposition to the goal of keeping the SD simple and straightforward.

3. Step-Single-Instruction Function and Instruction Pipeline

   The Step-Single-Instruction function causes many instructions to be executed as the result of one Step-Single-Instruction command. The machine's instruction pipeline is also flushed. On a broken machine, a possible side effect is that code may run differently depending on whether the Step-Single-Instruction function is being used.

4. Step-Over-This-Call function requires instruction decode knowledge

   The SD, in order to implement the Step-Over-This-Call function, requires instruction decode logic and knowledge of the calling standard. The same instruction, Jump-to-Subroutine (JSR), is used for both procedure calls and unconditional branches.

Executive Routines

This chapter summarizes the Mica executive routines.

CHAPTER 14

EXECUTIVE ROUTINES

14.1 Overview

This overview contains guidelines for designing Mica executive routines, and the protocol for submitting system service definitions to the author of the Internal System Services Manual.

NOTE

It is the intent of Mica to provide a stable platform, with respect to system services, that all user-mode applications can depend on. From time to time user-visible data structures and procedure interfaces will change. Mica intends to provide a system service interface such that user-mode applications do not have to re-compile or re-link to run on new versions of the operating system.

14.1.1 System Routines

System routines exist in three different forms:

1. User-visible, implemented in user-mode.
2. User-visible, implemented in kernel-mode.
3. User-invisible, implemented in kernel-mode.

This paper only discusses system routines that are implemented in kernel-mode. The name assigned to this class of system routines is *executive routines*.

System-routines implemented in kernel-mode that are visible in user-mode are known as *system services*. Mica system services have the facility name *exec$*.

System-routines implemented in kernel-mode that are invisible in user-mode are known as *executive services*. Mica executive services have many different facility names, with the most common facility name being *e$*.

The rest of this paper contains some code fragments and naming guidelines. The facility name used in all of the examples is *exec$*. This is for convenience only. When dealing with system services, the *exec$* facility name is correct, but for executive services the facility name is not *exec$*.

14.1.2 Executive Routine Interface Guidelines

This section describes the interface style for all executive routines in the Mica system. The intent is to provide a framework for the system service authors so that they can design system services that are similar in style to all other system services present in the Mica system, and to ensure that the styles used for system services are compatible with the style used in executive services.

All guidelines will be expressed in terms of Pillar. No guidelines for parameter passing technique i.e. pass by reference, value, descriptor will be given as this is a function of the Pillar language, and of the PRISM calling standard.

The guidelines are broken up into three areas:

- General Guidelines
- Object Service Routines
- General Service Routines

14.1.2.1 General Guidelines

This section describes the "prototype" executive routine. The important aspects are:

- Parameter Options
- Parameter Ordering
- Parameter Types
- Return Value

14.1.2.1.1 Parameter Options

The options for executive routine parameters should either be "required parameter", or **default**. The use of the Pillar **optional** parameter option is discouraged since these get passed by reference which is slow and needs to be probed and captured by the executive.

If a parameter is **default**, then it is assumed to be an optional parameter with respect to the caller of the executive routine. For the rest of this document, it is assumed that "optional" parameters use the **default** keyword.

14.1.2.1.2 Parameter Ordering

The parameter ordering for Mica executive routines is as follows:

- Required **IN** parameters.
- Optional **IN** parameters.
- Required **IN OUT** parameters.
- Optional **IN OUT** parameters.
- **OUT** parameters.

While the above parameter ordering scheme is encouraged, there are some instances where the rules are somewhat relaxed. The two primary reasons for using an alternate parameter ordering scheme are:

- The executive routine has a complex parameter list with a natural parameter grouping. In this case, the above scheme is enforced only for related groups of parameters.
- The executive routine is an object service routine. In this case, the object identifier is always the first parameter.

14.1.2.1.3 Parameter Types

Parameter types are always described in terms of named data types of the form *exec$t_xxxx* where *xxxx* is a reasonable description of the data type. The use of builtin Pillar data types is discouraged in the description of Mica executive routine parameters. The primary reason is to provide an easier migration path to the eventual 64 bit PRISM architecture.

The use of *item lists* is discouraged. The only endorsed use of item lists is in Mica executive routines that are used in configuring portions of the system or in the "get/set information" object service routines.

Where possible, parameters should be sized such that they will fit into a register.

14.1.2.1.3.1 Record Types

The use of Pillar records is discouraged since they must be captured and probed by the executive, and they will typically not be passed in registers. Executive routine designers should be aware that the use of records will significantly decrease performance. The use of records should be avoided for high bandwith executive routines.

NOTE

The rest of this section speaks in terms of executive routines, but it is really addressing executive routines that are system services. Executive services will always be compiled together. Record incompatibilities will not be an issue within the executive.

If an executive routine must use a record, then it must provide a mechanism where the record may change yet old code will still work properly. This allows for expansion without the use of item lists. The mechanism used to assure compatibility between exported records, and the executive routines that have record arguments is bound by the following goals:

- The caller of the executive routine should only ever see a single data type that describes the record.
- If the executive routine changes the definition of the record, but the caller does not recompile, then the call should still work and the executive routine may only use the previous version of the record. This strategy should hold for all versions of Mica. Only upward–compatible changes should be made to records.
- If the caller recompiles, then the new version of the record is in effect.
- The burden of determining which version of the record that the caller is using is placed on the executive routine.

NOTE

It is expected that Pillar will provide language support for detecting and converting versioned records. All executive code should be written in terms of up–to–date records. The method for automatic recognition, capturing, and converting versioned records is TBD.

14.1.2.1.4 Return Value

All Mica executive routines that return complex information, or are system services should return a value of type **status**. Executive routines that rarely fail, should report exceptional failures by raising conditions and should not return any values.

14.1.2.2 Object Service Routines

The guidelines expressed in Section 14.1.2.1 apply to object service routines with a few exceptions. As stated previously, the object identifier for the object service routine is always the first parameter. In addition to this deviation, others exist for the following classes of object service routines.

- Object Creation Executive Routines
- Get/Set Information Executive Routines

14.1.2.2.1 Object Creation Executive Routines

Object creation executive routines impose the following restrictions:

- The name of the executive routine is *exec$create_xxxx* where *xxxx* is the type of object being created.
- The first two parameters of the executive routine are:
  1. The object identifier of the object being created.
  2. The object type independent parameters of the object.
- The object type independent parameters of the object is described by the *exec$t_object_parameters*. This data structure contains the object name, the access control list for the object, and the container that the object is to be created in.

  Following the two required parameters are object type specific parameters. The ordering of these parameters is per Section 14.1.2.1.2.
- Object creation executive routines should simply create and initialize the new object. They should not perform other functions on the newly created object.

Example 14–1 illustrates a "prototype" object creation executive routine.

Example 14–1: Prototype Object Creation Routine

```
PROCEDURE exec$create_framitz (
            !
            ! Object Creation Required Parameters
            !
            OUT object_id: exec$t_object_id;                          ! returned id of the new framitz object
            IN  object_parameters: exec$t_object_parameters = DEFAULT;

            !
            ! Object type specific parameters
            !
            IN  framitz_state: exec$t_framitz_states;
            IN OUT framitz_sequence: exec$t_counter;
            OUT  framitz_length: exec$t_length;
            ) RETURNS status;
```

It is important to note that for the optional parameters, it is up to the designer of the executive routine to use proper values when actually creating the object. This also means that the executive routine may choose to ignore some of the optional parameters when appropriate.

Example 14–2: Prototype Get/Set Information Object Service Routines

```
!
! Prototype Get Information Object Service Routine
!
PROCEDURE exec$get_framitz_information (
            IN object_id: exec$t_object_id;          ! Object ID of framitz object
            IN information: exec$t_item_list;        ! Specification of information to be returned
            ) RETURNS status;

!
! Prototype Set Information Object Service Routine
!
PROCEDURE exec$set_framitz_information (
            IN object_id: exec$t_object_id;          ! Object ID of framitz object
            IN information: exec$t_item_list;        ! Specification of information to be set
            ) RETURNS status;
```

14.1.2.2.2 Get/Set Information Executive Routines

For each object type present in the Mica system, an interface capable of extracting or setting various attributes of the object should exist. While this is not a requirement, it is the preferred method of reading information from an object or configuring an object. The procedure name for the get /set information object service routines is in the form *exec$get_xxxxx_information()*, *exec$set_xxxxx_information()* where *xxxxx* is the name of the associated object type. When present, the format of the get/set information object service routines is as in Example 14–2.

The only acceptable deviation from the prototype interfaces described in Example 14–2 is to make the *object_id* parameter optional. This is only acceptable when the notion of "current" object makes sense. An example of this would be in a call to *exec$get_thread_information()*. The object identifier of the thread can be optional since the notion of "current thread" is reasonable.

It is important to note that item lists are present in the prototype get/set information object service routines. This is acceptable in these cases due to the relatively low performance requirements of this class of executive service, and the potentially large number of items.

14.1.2.3 General Executive Routines

There are no special rules that have not already been covered that apply to general executive routines. It is however the intent of the Mica system to provide simple interfaces that perform a single function with fairly well defined error modes. If at all possible, there should only be one way to perform a given function through the Mica executive routine interface. To illustrate this point, assume that for event objects there is a create interface and a wait interface. It would be inappropriate to add an option to the create interface that let the caller immediately wait on the object.

14.1.3 System Service Definitions

The designers of Mica system services are responsible for submitting the definitions for their system services to the author of Internal System Services Manual. The author of this chapter is currently Bill Muse.

System service definitions are submitted when the chapter identifying the system services has passed its formal review. The layout of a system service definition is based on the Pillar coding standards for external procedure definitions and the comment block for procedure implementations. Example 14–3 illustrates the proper layout of a system service definition for the *exec$set_thread_priority()* system service.

Example 14–3: Sample System Service Definition

```
!
! System Service Definition
!
PROCEDURE exec$set_thread_priority (
            IN thread_id: exec$t_object_id = DEFAULT;
            IN new_priority: exec$t_priority;
            OUT old_priority: exec$t_priority;
            ) RETURNS status;
            EXTERNAL;

!++
!
! Routine description:
!
!   Change the priority of the specified thread and return its old priority. The caller
!   must have write access to the thread.
!
! Arguments:
!
!   in thread_id        The object ID of the thread whose priority is to change. If the object ID has
!                       the default value, then the current thread is assumed
!   in new_priority     The new priority for the specified thread
!   out old_priority    The previous priority of the specified thread. This field is only valid if the
!                       return code is exec$c_normal
!
! Return value:
!
!   exec$k_normal       The priority change was performed
!   exec$k_no_access    Access to the specified thread was denied
!
!--
```

I/O

This set of chapters describes I/O components of Mica.

CHAPTER 15

DIRECT ACCESS MASS STORAGE FUNCTION PROCESSORS

15.1 Overview

This chapter describes the *logical block interface* to disks, which is used by function processors to access data residing on the disks. This chapter also describes the three function processors that implement the disk logical block interface:

- MSCP class function processor
- Striping function processor
- Shadowing function processor

Figure 15–1 shows the relationships between these function processors, the function processors they use, and function processors that use the disk logical block interface.

User-mode threads request disk I/O operations through the system services described in Chapter 8, I/O Architecture. Data transfer requests are always handled first by a file system function processor. The file system function processor uses the disk function processors to actually perform the data transfer.

Each of the *shadowing* and *striping* function processors implement and use the disk logical block interface. This allows stripe sets, shadow sets, and individual disk volumes to be transparently supported by file system function processors.

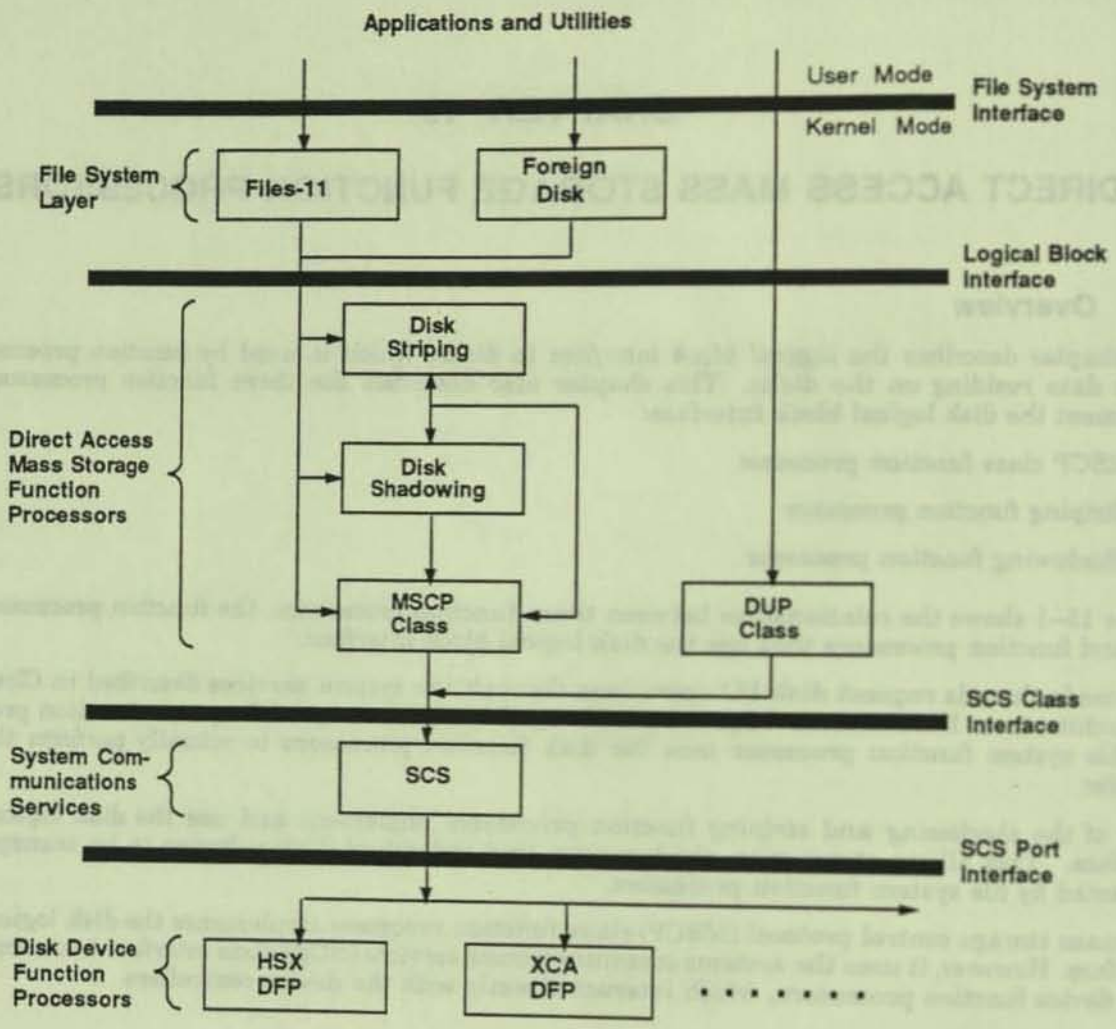
The mass storage control protocol (MSCP) class function processor implements the disk logical block interface. However, it uses the systems communications services (SCS) class interface to communicate with device function processors, which interact directly with the device controllers.

15.1.1 Goals

The objective of the disk logical block interface is to:

- Support all features of DSA 1 and DSA 2 disks
- Provide high throughput and low latency access to disk-resident data
- Provide features to support high availability and reliability of disk-resident data

Figure 15–1: Direct Access Mass Storage Function Processors and Clients



15.1.2 Disk Logical Block Interface

The disk logical block interface is a set of procedure calls that configure disks and access data on disks. The actual procedure calls are defined in Chapter 8, I/O Architecture. The function codes and parameter records that make up the remainder of the interface are described in this chapter.

The disk logical block interface supports reading, writing, comparing, erasing, and accessing data on disks. These operations are called *data transfer functions*, which are supported by all logical block unit function processors. The logical block interface also supports various *disk configuration functions*, which include bringing a unit online, initializing a stripe or shadow set, adding a counterpart to an existing shadow set, and so on. Most of the configuration functions are unique to a specific function processor.

15.1.3 Function Processors

The disk logical block interface is implemented by the MSCP function processor, the disk striping function processor, and the disk shadowing function processor.

15.1.3.1 MSCP Function Processor

The MSCP class function processor implements the logical block interface to disk device function processors. Each of its function processor unit (FPU) objects represents a disk unit.

The MSCP function processor's main purpose is to build MSCP requests, and queue them to the appropriate device function processors. It also manages changes in the physical configuration and generates error log packets.

15.1.3.2 Disk Striping Function Processor

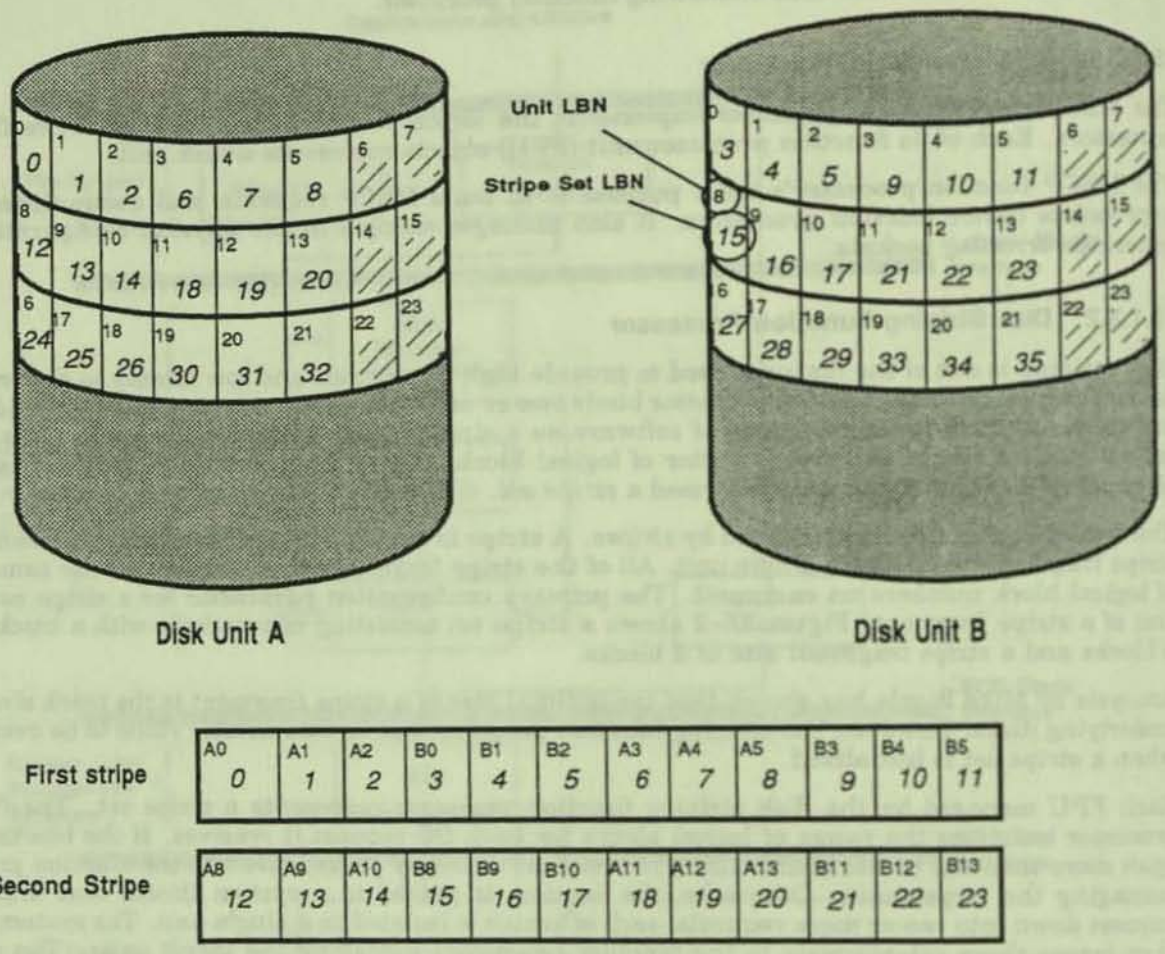
Disk striping is one of the features used to provide high throughput and low latency to disk-resident data. The disk striping function processor binds two or more homogenous logical block units together, and presents them to higher levels of software as a single, large, unified virtual unit. This unit is presented as a single continuous vector of logical blocks that is used exactly as if it were a single physical unit. This virtual unit is termed a *stripe set*.

The data in a stripe set is organized by *stripes*. A stripe is made up of a set of *stripe fragments*. Each stripe fragment resides on a single unit. All of the stripe fragments in a stripe cover the same range of logical block numbers on each unit. The primary configuration parameter for a stripe set is the size of a stripe fragment. Figure 15–2 shows a stripe set consisting of two disks with a track size of 8 blocks and a stripe fragment size of 3 blocks.

Analysis by Mike Riggle has shown that the optimal size of a stripe fragment is the track size of the underlying disks. However, the striping function processor allows this default value to be overridden when a stripe set is initialized.

Each FPU managed by the disk striping function processor represents a stripe set. The function processor examines the range of logical blocks for each I/O request it receives. If the blocks do not span more than one of the bound units, the request is simply passed down to the function processor managing the target unit. Otherwise, the request is passed to a system thread that breaks the request down into two or more requests, each of which is isolated to a single unit. The system thread then issues these sub-requests to the function processors managing the target units. The original I/O request does not complete until each of these sub-requests has completed. Similarly, the original I/O request's completion status is computed from the status values returned for the sub-requests.

Figure 15–2: Sample Stripe Set



15.1.3.3 Disk Shadowing Function Processor

Disk shadowing is a feature used to provide high data availability and enhanced reliability. The disk shadowing function processor binds two or more homogenous logical block units together to increase data reliability and availability. Each logical block unit contains a complete copy of the data on the others. If the data on one unit becomes unavailable, the data is still available on one of the others.

Each FPU managed by the disk shadowing function processor represents a *shadow set*. A shadow set consists of one or more *counterparts*. Each counterpart is a single logical block unit. Ideally, each counterpart holds exactly the same data as every other counterpart in the shadow set.

Disk shadowing has the potential to increase throughput and reduce latency for read operations, since the function processor can balance the load between the counterparts in the shadow set.

Much of the complexity of the shadowing function processor lies in error handling and recovery. The disk shadowing function processor will automatically and transparently replace damaged data (such as uncorrectable ECC error) on one counterpart with undamaged data from another counterpart. Counterparts are added to an existing shadow set while the shadow set is online. The new counterparts are automatically brought up to date with the data from the existing counterparts.

15.1.4 Error Handling and Diagnostics

15.1.4.1 Invalid I/O Request

An I/O request is invalid if any of the following is true:

- The target device (physical or logical) is not in an appropriate state to perform the requested action
- The I/O request does not contain a required parameter
- A field in the I/O request contains a value outside the domain of allowed values for that field
- The I/O request contains parameters that are not applicable to the requested action

Each function processor will detect the first three cases and terminate the I/O request with an error. The last case is very difficult to catch, and no attempts will be made to do so. Function processors simply ignore such fields.

15.1.4.2 Power Failure

Function processors are expected to gracefully recover from power failure. Function processor power recovery involves the following steps:

1. Terminating each outstanding I/O request with POWERFAIL or BAD_STATE status.

   When a function processor notices that an I/O request completed with POWERFAIL or BAD_STATE status, it needs to make a decision:

   - Allow the request to fail
   - Queue the request so it can be retried after the power recovery activity has completed

   Generally, only file systems have enough context to retry the request. Other disk function processors do not retry requests following powerfail recovery.

2. Returning all units back to their state before the failure.

   This is a bottom up process that involves reinitializing controllers, bringing units back on-line, re-establishing shadow and stripe sets, and performing mount verification at the file system level. The TRANSITION FPU state is provided by the I/O architecture to allow function processors to coordinate their powerfail recovery actions.

3. Retrying the requests that were requeued.

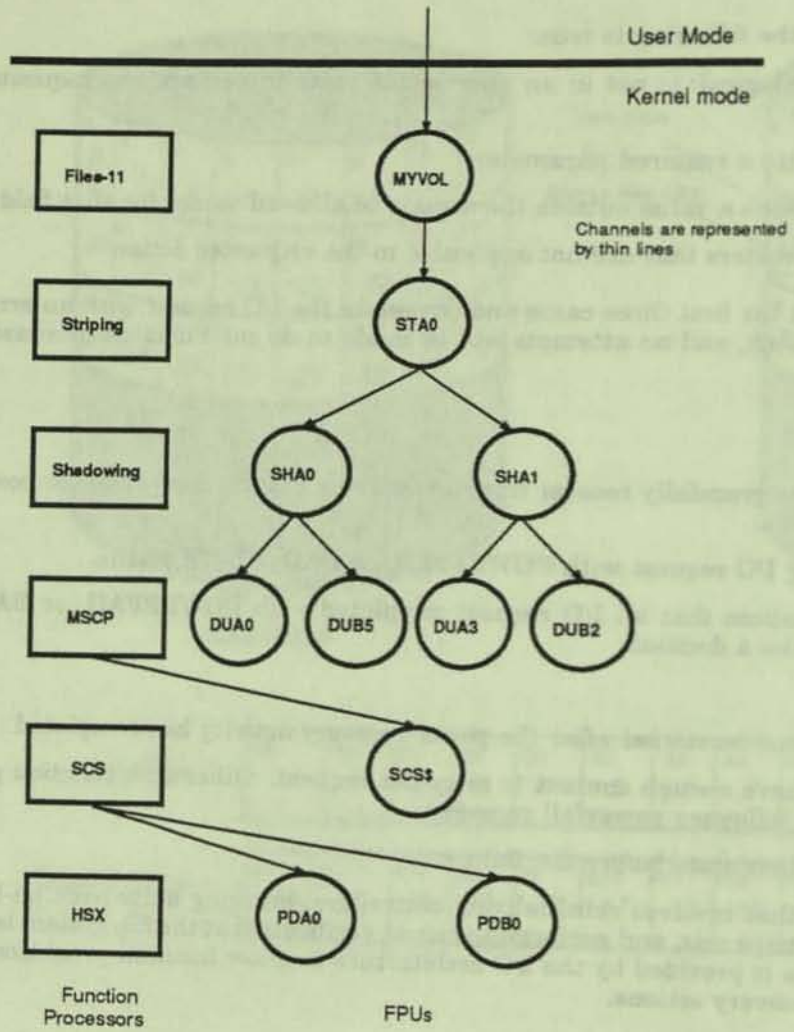
15.1.5 Sample I/O Request Flow

This section provides an overview of the flow of a disk I/O request through the I/O system. It should enable you to "get the feel" of how the various components fit together. The scenario is an application program that asks RMS to read a record from a disk file. The disk file resides on a stripe set, which is shadowed. Each shadow set has two counterparts, each of which is an RA90 disk unit. Both RA90s are connected to the host through an HSX controller. The stripe set fragment size is 69 sectors (the size of an RA90 track). Figure 15–3 shows the function processors, function processor units, and channels referenced in this example.

Figure 15–4 shows the I/O request packets (IRPs) used to satisfy the request. The following description details the actions that are taken during the execution of the request:

1. The application calls RMS at its $GET entry point

Figure 15–3: I/O Structure Layout for a Shadowed and Striped Files–11 Volume



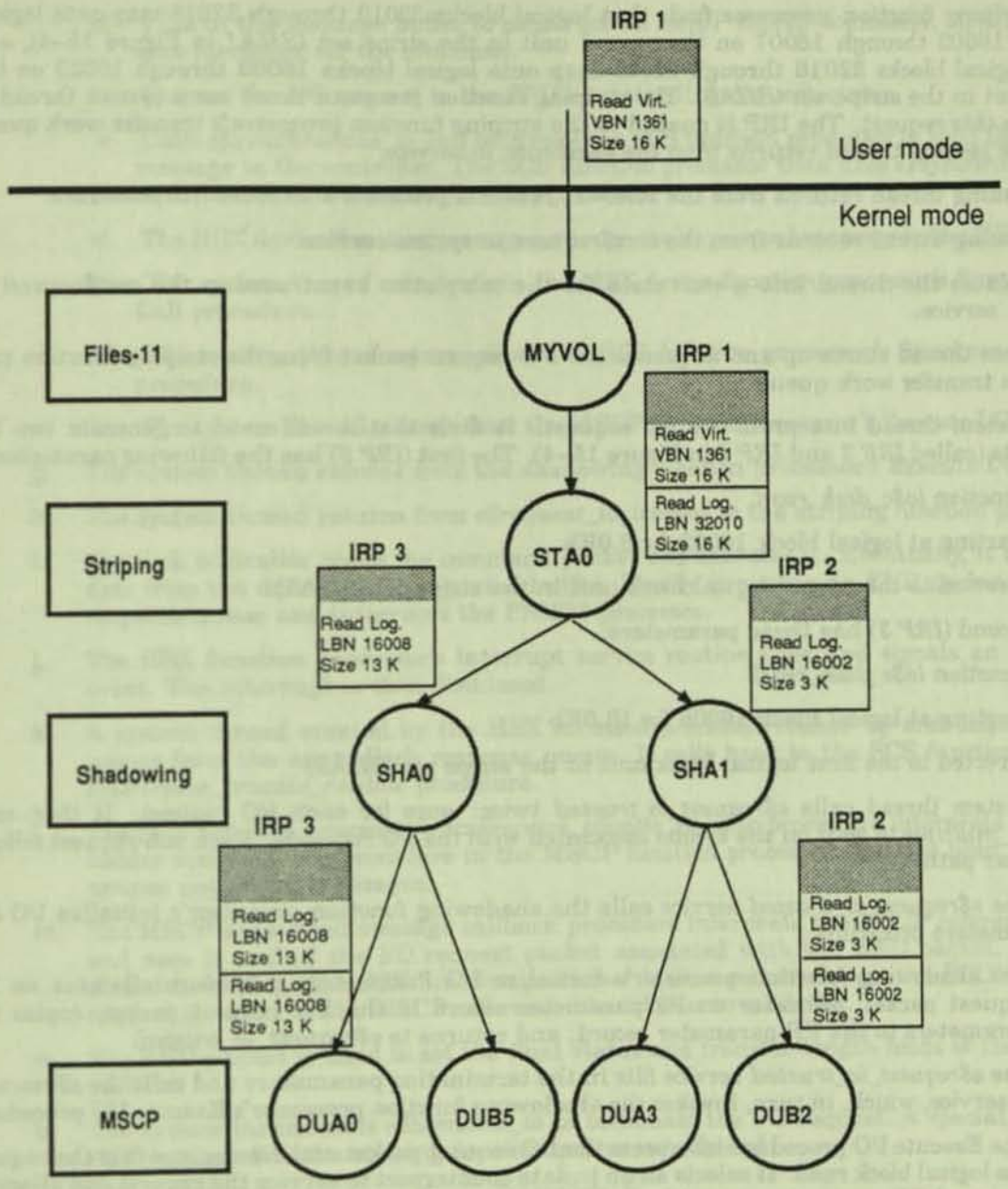
2. RMS determines that it must read the record from the file. RMS calls the *exec$request_io* system service, passing the object ID of the channel object on which it has previously accessed the file. The function code is *io$c_dfile_read*, starting at virtual block 1361, for 16 Kb.

3. The *exec$request_io* system service entry point dispatches to *e$request_io*, passing it the same parameters as RMS passed to *exec$request_io*, plus the access mode (user).

4. The *e$request_io* service then calls *e$request_io_trusted*, which finds the FPD dispatch table from the channel, and invokes the Files–11 function processor at its Initialize I/O Parameters procedure.

5. The Files–11 function processor's Initialize I/O Parameters procedure allocates an I/O request packet (marked *IRP 1* in Figure 15–4), interprets the parameter record, fills in the I/O request packet, and returns to *e$request_io_trusted*. The *e$request_io_trusted* service then fills in the termination parameters (IOSB, event, AST) and calls the *e$execute_io* service, which, in turn, invokes the function processor's Execute I/O procedure.

Figure 15–4: I/O Request Packets Used to Satisfy the Sample Request



6. The Files–11 Execute I/O procedure determines that virtual blocks 1361 through 1392 fall into a single file extent starting at logical block 32010. It allocates a second FP parameter record in the IRP, and places the following information into it:

   — *io$c_disk_read* function

   — Starting at logical block 32010 for 16 Kb

   The function processor calls *e$execute_io*, passing it the I/O request packet (*IRP 1*) and the address of a channel object created by the Files–11 function processor. The channel object identifies the FPU for the stripe set holding the file (*STA0* in Figure 15–4).

7. The I/O subsystem finds the FPD dispatch table from the channel and invokes the striping function processor's Execute I/O procedure.

8. The striping function processor finds that logical blocks 32010 through 32015 map onto logical blocks 16002 through 16007 on the second unit in the stripe set (*SHA1* in Figure 15–4), and that logical blocks 32016 through 32041 map onto logical blocks 16008 through 16033 on the first unit in the stripe set (*SHA0*). The striping function processor must use a system thread to process this request. The IRP is queued to the striping function processor's transfer work queue and the issuing thread returns from the *e$execute_io* service.

9. The issuing thread returns from the Files–11 function processor's Execute I/O procedure.

10. The issuing thread returns from the *exec$request_io* system service.

11. RMS places the thread into a wait state for the completion event used in the *exec$request_io* system service.

12. A system thread starts up and dequeues the I/O request packet from the striping function processor's transfer work queue.

13. The system thread interprets the I/O request. It finds that it will need to generate two I/O requests (called *IRP 2* and *IRP 3* in Figure 15–4). The first (*IRP 2*) has the following parameters:

    — Function *io$c_disk_read*

    — Starting at logical block 16002 for 3.0Kb

    — Directed to the second logical block unit in the stripe set (SHA1)

    The second (*IRP 3*) has these parameters:

    — Function *io$c_disk_read*

    — Starting at logical block 16008 for 13.0Kb

    — Directed to the first logical block unit in the stripe set (SHA0)

    The system thread calls *e$request_io_trusted* twice: once for each I/O request. It then calls *k$wait_multiple* to wait on the events associated with the I/O requests. Each sub-request follows a similar path:

    a. The *e$request_io_trusted* service calls the shadowing function processor's Initialize I/O Parameters procedure.

    b. The shadowing function processor's Initialize I/O Parameters procedure allocates an I/O request packet, allocates an FP parameter record in the I/O request packet, copies the parameters to the FP parameter record, and returns to *e$request_io_trusted*.

    c. The *e$request_io_trusted* service fills in the termination parameters and calls the *e$execute_io* service, which, in turn, invokes the shadowing function processor's Execute I/O procedure.

    d. The Execute I/O procedure interprets the I/O request packet and determines that the request is a logical block read. It selects an up to date counterpart to service the request and allocates a second FP parameter record in the I/O request packet. Next, the Execute I/O procedure copies the parameters from its original FP parameter record into the new FP parameter record. \The shadowing function processor has to keep its FP parameter record so that it can intercept errors and reissue the command.\ Finally, the shadowing function processor calls *e$execute_io* to pass the I/O request to the MSCP function processor.

    e. The *e$execute_io* service uses the channel passed to it to locate the MSCP function processor's FPD object. It then calls the MSCP function processor at its Execute I/O procedure.

    f. The Execute I/O procedure interprets the I/O request, and performs the following actions:

       i Calls *e$synchronous_io_call* to invoke the SCS function processor to allocate a sequenced message buffer.

ii Calls *e$synchronous_io_call* to invoke the SCS function processor to allocate a named buffer descriptor for the data buffer.

iii Fills in the message buffer to construct an MSCP READ command. Allocates a reference identifier for the command.

iv Queues the I/O request to the FPU's outstanding request queue.

v Calls *e$synchronous_io_call* to invoke the SCS function processor to send the sequenced message to the controller. The SCS function processor then uses *e$synchronous_io_call* to invoke the HSX device function processor.

vi The HSX device function processor queues the sequenced message to the HSX controller.

vii The system thread returns from the HSX device function processor's Synchronous I/O Call procedure.

viii The system thread returns from the SCS function processor's Synchronous I/O Call procedure.

ix The system thread returns from the MSCP function processor's Execute I/O procedure.

g. The system thread returns from the shadowing function processor's Execute I/O procedure.

h. The system thread returns from *e$request_io_trusted* to the striping function processor.

i. The disk controller reads the command packet and executes it. Eventually, it transfers the data from the disk to the designated buffer. Finally, it queues an END packet to the port's response queue and interrupts the PRISM processor.

j. The HSX function processor's interrupt service routine runs and signals an autoclearing event. The interrupt is then dismissed.

k. A system thread created by the HSX function processor wakes up and dequeues the response from the controller's response queue. It calls back to the SCS function processor's *scs$receive_process_routine* procedure.

l. The SCS function processor's *scs$receive_process_routine* procedure interprets the message header and calls the procedure in the MSCP function processor that has been registered to process sequenced messages.

m. The MSCP sequenced message callback procedure interprets the MSCP reference identifier and uses it to find the I/O request packet associated with the END packet. The system thread created by the HSX function processor is then removed from the FPU's outstanding request queue, and the MSCP reference identifier is invalidated.

n. The END packet is used to set the final status and transfer length fields of the I/O request packet.

o. The system thread calls *e$complete_io* to terminate the I/O request. A special kernel mode AST is queued to the striping function processor's thread.

p. The system thread returns from *e$complete_io*.

q. The system thread returns from the MSCP function processor's sequenced message callback procedure.

r. The system thread returns from the SCS function processor's *scs$receive_process_routine* procedure.

s. The system thread checks for another processor response, and waits if there is none.

t. The special kernel-mode AST is delivered to the striping function processor thread. The AST procedure terminates the processing for the sub-request (*IRP 2* or *IRP 3*) and deletes the I/O request packet. The AST is then terminated.

14. The striping function processor's system thread becomes ready to run when the last of its subrequests have completed.

15. The system thread computes the final status, and calls *e$complete_io* to terminate the I/O request (*IRP 1*).

16. The system thread goes back to waiting for work on the striping function processor's work queue.

17. A special kernel-mode AST is delivered to the process thread that issued the original I/O request. The AST procedure writes the I/O status block, then signals the completion event. The I/O request packet (*IRP 1*) is deleted, and the AST is terminated.

18. The process thread resumes running in user mode and executes RMS code to check the completion status of its I/O request. RMS deblocks the record from the buffer and presents it to the caller.

19. The process thread returns back to the caller from RMS.

CHAPTER 16

MAGNETIC TAPE FUNCTION PROCESSORS

16.1 Overview

This chapter describes the support for magnetic tapes in Mica. Mica provides access to unlabeled, foreign-mounted magetic tapes. While the initial product set does not include ANSI-tape support, future versions of the software probably will include such support.

The following interface and function processor are described in this chapter:

- Tape logical-block interface
- *Tape Mass Storage Control Protocol (TMSCP)* function processor

Figure 16–1 shows the relationship between the TMSCP function processor and other function processors it uses.

The TMSCP function processor implements the Tape Mass Storage Control Protocol. Threads request tape I/O operations through the system services described in Chapter 8, I/O Architecture. The TMSCP function processor supports requests from kernel- and user-mode threads.

16.1.1 Goals and Requirements

Mica tape support meets the following requirements:

- Support all DSA-1 and DSA-2 TMSCP-compliant storage elements supported by HSX controllers, HSC controllers, or both

  TMSCP storage elements that are not supported by these controllers are not supported by Mica. Although little or no development would need to take place to support them, they are not included in system testing.

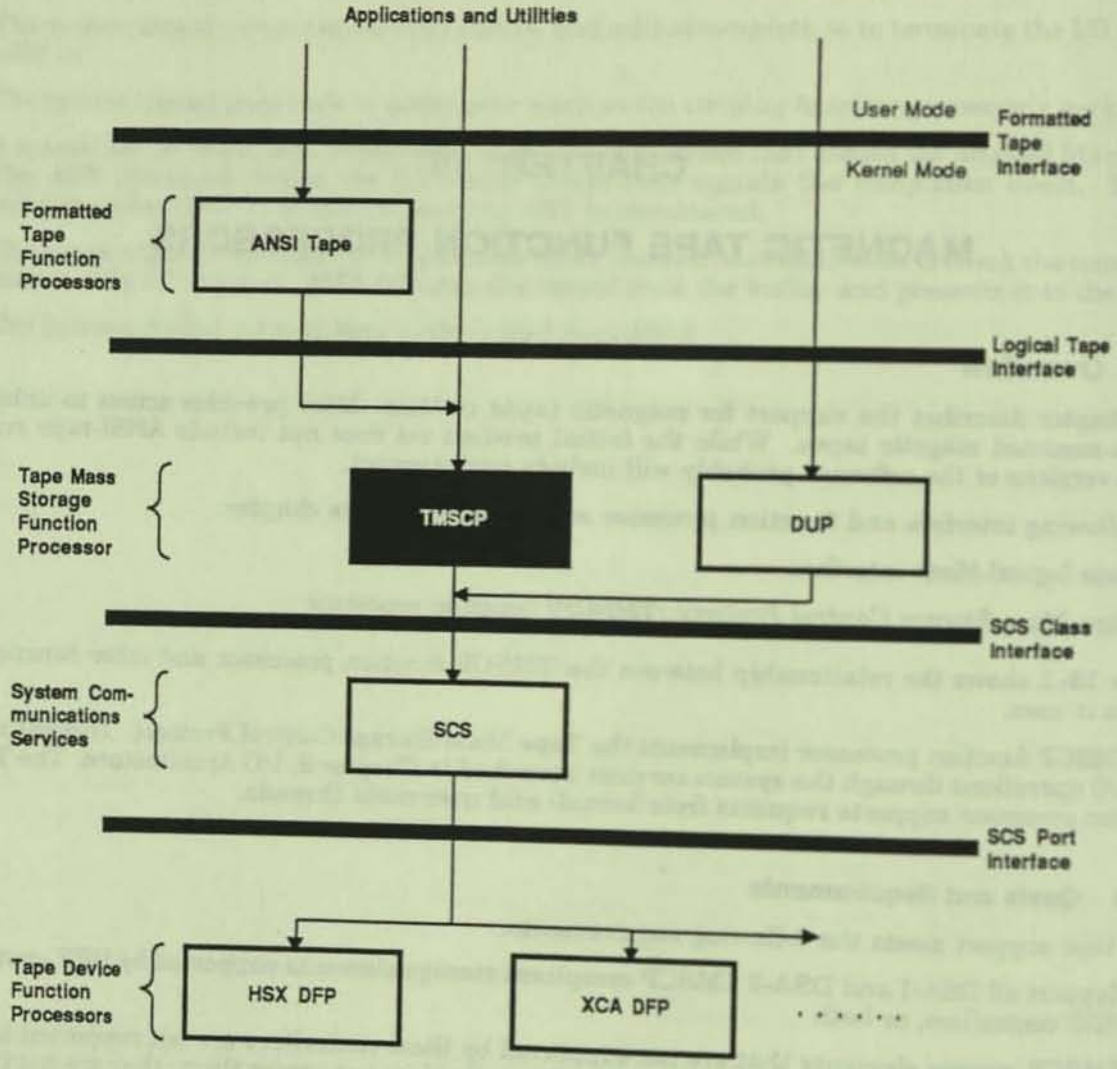
- Allow streaming tape drives to run at their maximum data rates
- Support the Ptolemy optical-storage device if required by Glacier or Cheyenne

  The Ptolemy optical disk is a write-once, TMSCP-compliant storage device. Supporting Ptolemy is limited to including it in the system test plans, and providing support in the error-log display program. The optional media loader is supported in its transparent, automatic mode, but not in its program-control mode. This support allows a stack of media to be loaded into the drive and processed in FIFO order.

- Support exclusive access to tape drives

  A tape drive is only allowed to be ONLINE through a single host. AVAILABLE tape drives are allowed to be AVAILABLE to any host. This allows a pool of tape drives to be shared among several hosts, but limits the processing of data to one host for each drive.

Figure 16–1: Magnetic Tape Function Processors



16.1.2 Tape Logical-Block Interface

The tape logical-block interface is a set of procedure calls used to configure tapes and access data on tapes. The procedure calls are defined in Chapter 8, I/O Architecture. The function codes and parameter records that make up the remainder of the interface are described in this chapter.

The tape logical-block interface supports reading, writing, comparing, erasing, accessing, and positioning-to data on tapes. These functions are called *data-transfer functions*. It also supports various *tape-configuration functions*. These include bringing a unit on line, setting physical parameters governing the media (such as density, speed, caching mode), and so on.

The tape logical-block interface does not support any form of labeled tapes. The interface merely provides the ability to access data on TMSCP-compliant storage devices; interpreting this data is the responsibility of higher layers of software.

TMSCP supports cached tape drives. The tape logical-block interface allows applications to choose cached or uncached operation for data-transfer requests that modify the media. The use of caching for other data-transfer operations is transparent to the function processor, and thus cannot be selected.

Uncached write operations do not complete until the data is on the media. Cached operations complete when the controller has read the data from the host, but possibly before the data is on the media. The use of caching dramatically improves throughput (by up to a factor of ten for some drives) at the expense of reliability. Reliability suffers because applications may be told their I/O request has completed before the data is actually recorded on the media. The tape logical-block interface provides mechanisms whereby applications can enable write caching, but still synchronize with the media when necessary (for example, by writing data with the *io$c_item_caching* flag set to FALSE, or by using the *io$c_tape_flush* function).

16.1.3 TMSCP Class Function Processor

The TMSCP class function processor implements the tape logical-block interface. Each TMSCP function processor unit (FPU) represents a tape drive.

The TMSCP class function processor's primary purpose is to translate application I/O requests into their TMSCP counterparts. It also manages the configuration of TMSCP tape drives, and performs error logging for TMSCP tape drives.

Data-transfer requests are processed in the context of the thread issuing the request. The TMSCP function processor communicates with device function processors through the *System Communication Services (SCS)* function processor. End-packet processing is performed in the context of device function processor threads.

The TMSCP function processor creates one thread to manage each tape controller. This thread is used to configure units on the controller, and for access-path failover and powerfail-recovery processing.

Much of the design and implementation of the TMSCP function processor is derived from the *Mass Storage Control Protocol (MSCP)* class function processor, described in Chapter 15, Direct Access Mass Storage Function Processors.

CHAPTER 17

SYSTEM COMMUNICATION SERVICES

17.1 Overview

This chapter specifies the System Communication Services (SCS) available in the Mica system. It also outlines the SCS implementation strategies identified to date.

Please see the attached glossary for a short list of SCS terms and definitions. In this chapter, *port* is used to refer to a remote node's interface to the Computer Interconnect (CI), bus and *adapter* refers to the local Mica CI interface, namely the External Memory Interconnect (XMI)-to-CI Adapter (XCA). Any local Mica *controllers*, such as the Hierarchical Storage Controller for XMI (HSX), using *System Communication Architecture* (SCA) are treated as adapters by SCS.

17.1.1 Goals and Requirements

The objective of the Mica SCS interface is to:

- Conform to the SCA specification for message formats, protocol, and implementation-independent interface models
- Provide SCA communication services for concurrent Mica system applications running in kernel mode to remote computer systems over one or more multi-access interconnects

Functions not implemented in the Mica SCS interface are:

- Failover support. SCS reports the failure of a communication *path* to the Mica system applications, but the individual application is responsible for reinitiating contact to a remote partner over another path.
- Management of load balancing between many paths to a single, remote system.
- The current implementation of SCA in CI port hardware can not guarantee that a sequenced message is always delivered or that the sender is always notified of an error. Consequently, the Mica implementation of SCS limits its efforts in this area. The design of system application software should take this limitation into account.

17.1.2 SCS Functionality

SCS performs the functions of the *session layer* of the DEC System Communication Architecture (SCA). As such, SCS is responsible for managing *connections* between communicating system applications (*SYSAPs*), controlling the use of buffer resources, controlling the flow of messages and data, and multiplexing different connections onto the *virtual circuits* between systems.

SCS provides user applications with the following services:

- Directory services

These services maintain a directory of local SYSAPs that are waiting for active connection requests from remote partners. A local SYSAP uses the services to register or delete itself from the directory. A remote user cannot directly access the directory, but rather communicates with a local Mica SYSAP, named *scs$directory*. The *scs$directory* SYSAP uses the directory services to provide directory lookup functions for remote users.

- Configuration services

  These services maintain the identity of remote systems reachable through SCS. They keep information about the remote systems' hardware and software, and the connecting paths to them. The local SYSAP uses these services to determine which remote systems are reachable, the number of paths to the remote system, and the state of the virtual circuit on each path.

- Connection services

  These services are used to establish a connection between two SYSAPs over a virtual circuit. Connection services give SYSAPs the capability to: a) request a connection to another SYSAP, b) accept or reject a connection request, c) disconnect from an established connection, or d) be notified when a connection is broken.

- Data exchange services

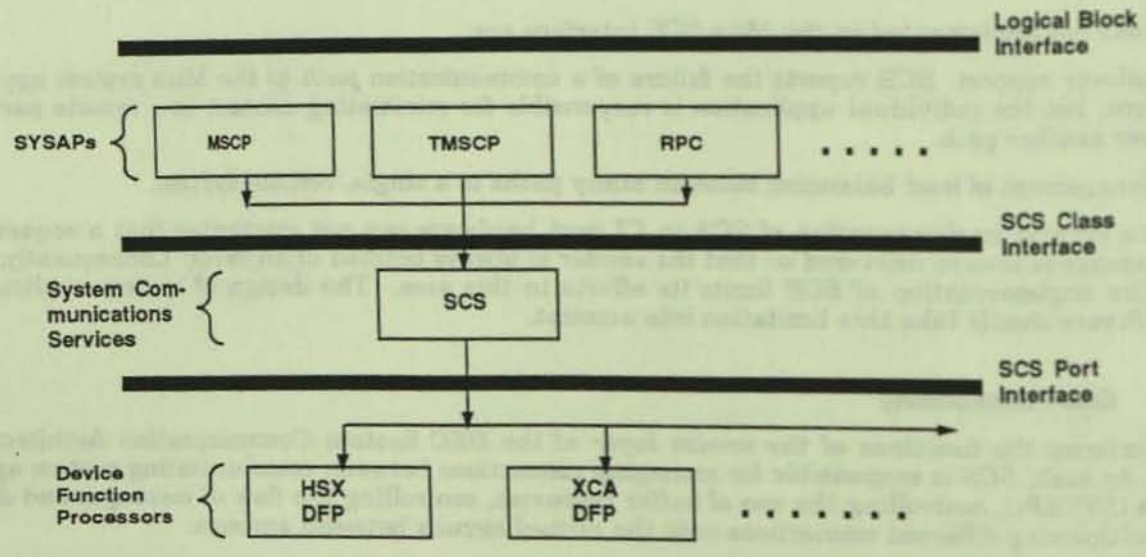
  Data exchange services allow two connected SYSAPs to exchange information. These services send and receive *datagrams*, *sequenced messages*, and block data transactions.

17.1.3 Implementation Strategy

17.1.3.1 Initialization and System/Path Recognition

SCS is implemented as a single function processor with a single function processor unit (FPU). Figure 17–1 shows the relationship between the SCS function processor and other function processors in the system.

Figure 17–1: SCS Function Processor in the I/O System



The SCS function processor is initialized when the configuration function processor recognizes the first adapter that uses SCA protocol. Thereafter, the configuration function processor informs SCS about every SCA adapter it successfully configures (see Chapter 34, Configuration Management Software). SCS responds by establishing a channel to the adapter FPU, and by binding the SCS FPU

to the adapter FPU. The adapter's device function processor finds the remote systems with which the adapter can communicate, establishes virtual circuits to each system, and records the resulting system information in the SCS configuration database.

Local SYSAPs can use the SCS directory and configuration services as soon as SCS is initialized. Once a system path is established and the virtual circuit is open, then a SYSAP uses SCS's connection services to establish a connection between itself and a SYSAP in a remote system. Thereafter, the SYSAP uses SCS data exchange services to communicate with the partner.

17.1.3.2 Message and Datagram Buffer Allocation

One of the goals of SCS is to make it unnecessary to copy application data from buffer to buffer as each software layer adds header information. Therefore, SYSAPs are required to obtain message and datagram buffers from SCS. SCS acquires these buffers from device function processors. A device function processor allocates a physically contiguous buffer that is big enough to hold the application data, along with the SCS header and the device function processor header.

SYSAPs cannot assume anything about the size or content of the SCS and device function processor layer headers. On the return from buffer allocation, the SYSAP has only a pointer to the beginning of the application data area.

17.1.3.3 SYSAP-SCS Interface

SCS implements the following request I/O function codes:

- *connect* establishes an SCA connection to a partner SYSAP. SCS allocates a specified number of message buffers (initial message credits) and queues them to the adapter's *message free queue* (MFREEQ). Then, a dialog to establish the connection to the partner SYSAP begins. The I/O request completes when either: a) the connection is established, b) the connection is rejected by the partner SYSAP, c) the target system becomes inaccessible through the identified path, or d) the partner SYSAP is not listening for a connect request.
- *accept* accepts a connect request from a partner SYSAP.
- *reject* rejects a connect request.
- *disconnect* disconnects a connection to a partner SYSAP.
- *read_block_data* transfers a block of data from the partner system to a local buffer.
- *write_block_data* transfers a block of data from a local buffer to a partner system.

SCS implements the following synchronous I/O entry points:

- *read_directory* searches the local SYSAP directory and returns the information about a specified SYSAP name or entry number stored there.
- *system_configuration* searches for information about accessible systems, given a system name or entry number.
- *path_configuration* returns information about an available system path.
- *enable_listen_ast* registers a SYSAP in the local directory and requests an AST for each connect request. If the SYSAP is unwilling to listen for additional connect requests, it calls this function with a null AST procedure address.
- *enable_disconnect_ast* establishes the AST procedure to be invoked if the connection to a partner SYSAP transitions from the OPEN state. This AST procedure is not invoked if the local SYSAP initiates a disconnect.
- *allocate_datagram* allocates a datagram buffer from the *datagram free queue* (DFREEQ). The SYSAP thread waits until the datagram can be allocated.

- *send_datagram* sends a datagram. The datagram buffer is put on DFREEQ after the datagram is sent.
- *deallocate_datagram* returns a datagram to the DFREEQ.
- *allocate_message* allocates an application message buffer from non-paged pool. The current thread waits until the message buffer is allocated.
- *send_message* sends an application-sequenced message. After the message is sent, the buffer is deallocated or placed on the MFREEQ, depending on the caller's wishes.
- *add_messages* adds message buffers to receive messages from remote partners. The function terminates when the buffers have actually been added. SCS must coordinate with the partner system when adding buffers.
- *deallocate_message* returns an application message buffer back to non-paged pool or the MFREEQ.
- *map_data_buffer* prepares a buffer for *block data transfer* by initializing a buffer descriptor in the buffer descriptor table (BDT).
- *unmap_data_buffer* releases a buffer descriptor for reuse.

17.1.3.4 SCS-Device Function Processor Interface

SCS and device function processors communicate with each other by synchronous I/O calls and SCS callback routines. The device function processors use SCS callback routines to announce data reception.

The SCS data reception callback function processes the received data to completion in the context of the device function processor thread. SCS determines the type of data received, and proceeds as follows:

- Datagrams—SCS delivers the datagram to the SYSAP via a callback routine. In most cases, the SYSAP processes the data immediately, or copies the data from the buffer for future processing. The SYSAP then returns the datagram buffer, through SCS and the device function processor, to the DFREEQ. However, the SYSAP has the option to keep the datagram buffer.
- Sequenced application message—SCS delivers the message to the SYSAP via a callback routine. In most cases, the SYSAP processes the data immediately, or copies the data from the buffer for future processing. The SYSAP then returns the message buffer, through SCS and the device function processor, to the MFREEQ. However, the SYSAP has the option to keep the message buffer.
- Block data transfers—SCS locates the IRP and completes the I/O request associated with the transfer.
- SCS protocol messages—The processing of protocol messages can cause an I/O request, such as a SYSAP-initiated *connect*, to complete, if the message being processed concludes protocol communication.

17.1.3.5 Flow Control Scheme

Flow control is used to ensure that a sender does not send more data than the receiver has buffer space. SCA requires flow control for sequenced messages and block data transfers, but not for datagrams.

Block data transfer is governed by flow control because SCA assumes that a message buffer is needed by the remote system to complete the transfer.

17.1.3.5.1 SCS Protocol Messages

SCS protocol messages are also sequenced messages that require flow control. Flow control for SCS protocol messages is transparent to SYSAPs, and is applied to paths, rather than to connections. When the virtual circuit for a path is opened, SCS sets aside one buffer for sending protocol requests and one buffer for receiving protocol requests. SCS then uses internal mechanisms to ensure that a simple request/response protocol is maintained. In this way, no new SCS request can be sent on a path until a response to the previous request on that path is received.

17.1.3.5.2 SCS Application Messages and Block Data Transfers

For application-sequenced messages and block data transfers, SCS maintains flow control for each SYSAP-to-SYSAP connection by means of a *credit* scheme. That is, SCS keeps track of the credits for each connection and does not allow a SYSAP to send any messages or block data transfers without the proper number of "send credits."

When a SYSAP establishes a connection with a partner, it specifies an argument called "initial credits." This is the partner's initial number of send credits and determines how many message buffers SCS must allocate and queue to the MFREEQ for this connection. During connection initialization, this send credit value is reported to the partner SYSAP.

Communicating credit updates to the remote partner has two parts:

- First, all sequenced message headers have a credit field. Using this field, SCS can piggyback credit information with each outgoing message. As receive buffers are added for a connection, a pending receive count is maintained. Whenever a message is sent out on that connection, the pending receive count is copied to the message header credit field, and then zeroed. As messages arrive at the remote system, the remote SYSAP's send credit is increased by the amount of the credit field.
- Second, SCS has a special credit protocol message available to it. SCS always monitors the remote SYSAP's send credit value. If the send credit reaches a specified threshold and the pending receive count is not zero, SCS uses the credit protocol message to notify the remote SYSAP of the additional send credits, and then zeros the pending receive count.

This implementation only allows send credits to be added. Any attempt to delete send credits results in protocol errors. SYSAPs can add send credits by using the *add_messages* interface, by sending an application message with its recycle flag set, or by returning a received message buffer with its recycle flag set.

17.1.3.6 Error Philosophy

Three types of errors are handled by Mica SCS:

- Invalid I/O requests

  If SCS receives an invalid I/O request, it fails the request with the appropriate error status.

- Adapter errors that result in the loss of the virtual circuit

  Adapter errors that result in the loss of the virtual circuit include adapter failure, host powerfail, and fatal adapter errors that cause adapter reinitialization.

  Recovery from an adapter error requires the cooperation of the device function processor, SCS, and the local SYSAPs. The device function processor is the first to recognize the event. The device function processor must:

  1. Complete all threads waiting on adapter resources with an error
  2. Notify SCS of the error via the device FPU state change AST
  3. Begin adapter recovery from the event

The SCS system thread takes the following actions:

1. Marks the virtual circuit as CLOSED for each path through this adapter
2. Terminates all I/O in progress on the connections through this adapter
3. Invalidates all connections for paths through the adapter
4. Contacts previously connected SYSAPs via their disconnect AST

The SYSAPs are responsible for reestablishing connections to their partners and resuming normal activity using SCS configuration services.

If adapter recovery was unsuccessful, or if SYSAPs do not resume normal activities, the SYSAPs are then responsible for deallocating all of the resources they hold for the discontinued connections.

- SCS protocol errors

SCS interprets an invalid request by a remote SYSAP as a protocol error and assumes that the remote SCS is broken, even though the adapter is considered sane and the device function processor is unaware of any problem.

Upon recognition of a protocol error involving a remote node, SCS queues the error to a system thread, which contacts SYSAPs using their disconnect AST.

CHAPTER 18

XCA FUNCTION PROCESSOR

18.1 Overview

18.1.1 Introduction

The XMI-to-CI Adapter (XCA) port is an intelligent controller that connects the External Memory Interconnect (XMI) bus to the high speed serial Computer Interconnect (CI). This chapter describes the XCA device function processor, which is the lowest-level Mica interface to the XCA device.

18.1.2 Requirements

The XCA device function processor is required to:

- Provide CI support for Mica
- Support multiple CI controllers
- Provide remote system recognition to ensure a current system configuration
- Provide port-to-port virtual circuit service using the system communication service (SCS) handshake protocol
- Deliver sequenced messages and block data in the correct order and without duplications
- Provide hooks for diagnostics, as needed

18.1.3 Goals

XCA device function processor goals are to:

- Isolate SCS from the XCA controller hardware
- Provide a communication medium that is free of undetected transmission errors
- Manage network congestion control by selected use of paths on dual-path CI hardware and prioritized controller queues

18.1.4 Functionality

The XCA function processor performs the following device-dependent functions for Mica:

- Initializes the XCA controller
- Maintains the current controller status for inquiry by higher-level function processors
- Manages memory data structures used for the Mica/controller interface
- Handles controller malfunctions and recovery
- Sends information packets (datagrams, sequenced messages, block data) to remote systems
- Receives information packets from remote systems and delivers each packet to its corresponding higher-level function processor
- Maintains raw data for system performance measurement
- Polls the controller at prescribed intervals to update system configuration data
- Opens and closes virtual circuits to remote systems

18.1.5 Higher-level Interface to XCA Function Processor

The significant functions provided by the XCA function processor for other function processors wishing to communicate with the XCA controller are:

- SCS port driver (PD) interface functions

  The SCS PD interface provides a consistent set of services to the SCS function processor from each device controller function processor. The XCA function processor implements the SCS PD interface described in Chapter 17, System Communication Services.

  The following is a summary of the available PD services:

  — Allocate datagram and message buffers

  — Release datagram and message buffers

  — Map and unmap block data buffers

  — Send datagrams, messages, and block data

  — Receive datagrams, messages, and block data

  — Open virtual circuits

  — Close virtual circuits

- Controller configuration functions

  The following XCA function processor procedure and function codes control the configuration of the XCA controllers:

  — Initialize FPU procedure—Initializes the function processor unit (FPU) for an XCA controller and sets the initial FPU state from the controller's status register

  — *io$c_xca_ready_fpu*—Initializes the XCA controller and sets the FPU state to ONLINE

  — *io$c_xca_poll_controller*—Forces the function processor to poll the controller's station addresses to define the system configuration

  — *io$c_xca_unready_fpu*—Changes the FPU state to AVAILABLE and resets the controller hardware and all controller state data in the FPU

Readying an XCA controller causes the function processor to begin system configuration polling. The *io$c_xca_poll_controller* function is used to configure remote systems without waiting for the normal polling time interval to elapse.

- Diagnostic functions

  The diagnostic capabilities of the XCA function processor will be specified when the requirements and functions are defined.

18.1.6 XCA Function Processor Interface to the XCA Port

The XCA function processor communicates with the XCA controller through:

- Port registers

  The port registers form the low-level path through which basic control and status operations are performed. The XCA function processor uses these 24 registers to:

  — Initialize the XCA port

  — Get basic controller status information

  — Control diagnostic operations

  — Force command queue reading by the controller

- Command and response queues

  Command and response queues control the normal operation of the controller. Port commands and responses are contained in blocks of host memory called *queue packets* and linked to an appropriate command or response queue. Each queue packet corresponds to a datagram or sequenced message with a standard header that defines the nature and parameters of the command or response, followed by the data particular to that command or response. Queue packets that are not in use have undefined contents and reside on free queues from which they can be obtained as needed.

  The controller is linked to the command and response queues by a data structure called the port queue block (PQB). The XCA function processor has one response queue, four command queues, a message free queue (MFREEQ), and a datagram free queue (DFREEQ).

  As the result of controller restrictions, the PQB, command and response queue headers, and queue packets are located in the low 512 MB of physical memory. Each individual PQB, queue header, and queue packet must be quadword aligned and physically contiguous within itself. The PQB must also be aligned on a 512-byte boundary.

- Buffer descriptors and host transfer lists

  When data movement involves block data transfers, then the host must supply the controller with buffer descriptors to define the block data buffers. The buffer descriptor is an entry in a buffer descriptor table (BDT), which is located using the PQB. The buffer descriptor points to a host transfer list (HTL), which is a list of the segments of memory that together constitute the buffer. The controller accesses the descriptor, determines the physical memory addresses of the memory segments, and accesses the block data buffer directly.

  The BDT and HTL must each be physically contiguous, quadword aligned, and located in the low 512 MB of physical memory.

- Interrupt service routines

  The XCA function processor uses two interrupt service routines. These routines are connected to the appropriate vectors for the XCA controller.

  — IPL 4 ISR—This interrupt service routine is invoked when the XCA controller inserts a response entry on an empty response queue. The service routine signals the XCA function processor with a pending response event (see Section 18.1.7.3).

— IPL 5 ISR—This interrupt service routine is invoked when the XCA controller declares an error, or writes its status register as the result of a state change. The service routine sets the processor's controller state change event (see Section 18.1.7.4).

18.1.7 XCA Function Processor Implementation

The Mica interface to all XCA controllers is implemented as a single function processor. The XCA function processor has one or more FPUs, each of which represents an XCA controller.

For each FPU created, the XCA function processor creates one main system thread and several worker threads. The worker threads are responsible for dequeuing packets from the controller response queue. The main system thread handles all other functions within the function processor.

All SCS PD-defined entry points to the XCA function processor are function codes for the *e$synchronous_io_call* service that execute in the context of the caller.

The XCA function processor passes received messages and datagrams back to a higher-level function processor through previously established response callback procedures. The threads executing the callback routines operate in the context of the XCA function processor.

18.1.7.1 System Recognition

SCS relies on an internal database to determine the system configuration. It is the responsibility of the XCA function processor to determine which remote systems each XCA controller can reach and include this information in the SCS configuration database. To accomplish this, the XCA function processor's main system thread periodically polls all 224 controller station addresses, sending request ID (REQID) packets to each station. The corresponding ID received (IDREC) packets, and any unsolicited IDREC packets, are then used to configure the system list.

18.1.7.2 Virtual Circuits

The XCA function processor provides virtual circuits for SCS. An open virtual circuit is needed for datagrams, messages, and block data to be exchanged between systems.

The XCA function processor's main system thread opens virtual circuits using a standard 3-way handshake. During system recognition activities, the XCA function processor automatically begins the open virtual circuit sequence when it discovers a path to a remote system that is not in the configuration database. Virtual circuit initialization also begins when the XCA function processor receives a start virtual circuit datagram from a remote system, or a call to its Synchronous_io_call procedure with an *io$c_xca_open_vc* function code.

The XCA function processor closes a virtual circuit when it notices a virtual circuit failure, or when a higher-level function processor requests a virtual circuit closure. The XCA function processor can detect virtual circuit failure during polling activities or data transmissions. The function processor handles such a failure by issuing a command to disable the virtual circuit in the controller's internal virtual circuit table and signaling the virtual circuit error AST.

An example of a higher-level function processor requesting a virtual circuit closure is SCS detecting a virtual circuit failure by finding protocol errors in communications with remote systems. SCS then issues a call to the XCA function processor's Synchronous_io_call procedure with an *io$c_xca_close_vc* function code. The XCA function processor updates the controller's internal virtual circuit table to disable the virtual circuit on this path, then returns to SCS.

The system and path information in the SCS configuration database, as well as the controller's internal virtual circuit table, are always updated with the results of the virtual circuit initialization or closure.

18.1.7.3 Response Handling

An event is signaled by the interrupt service routine whenever an entry is put on the response queue. An XCA function processor worker thread that was waiting on the event increments the active request count, and removes a response entry from the response queue. If the queue is still non-empty, the thread signals the pending response event again. The thread then decodes the response packet's type to decide on the necessary processing. If a higher-level function processor should receive the packet, then that function processor's response callback procedure is called. Upon returning from the callback routine, the system thread decrements the active request count, and loops back to wait for another pending response event.

18.1.7.4 Error Handling

The XCA function processor is notified of fatal errors when its main system thread's wait is satisfied by a controller_state_change event or a powerfail event.

For the benefit of higher-level function processors, the XCA function processor performs the following actions:

1. Sets the XCA's FPU state to TRANSITION (and increments the FPU's sequence number)
2. Completes all threads waiting on controller resources with an error
3. Records the controller's status and error values in an error log packet
4. Begins controller reinitialization actions to recover from the event

No matter what the outcome is of the recovery processing, the XCA function processor guarantees that the FPU state will not remain in the TRANSITION state indefinitely.

CHAPTER 19

NI FUNCTION PROCESSOR

19.1 Overview

This overview summarizes plans for the Mica NI function processor, a device function processor providing data link communication between NI (Network Interconnect) devices. Network function processors call the NI function processor to read from a NI communication device, or to write to such a device. For the rest of this document, the term *NI* refers to Ethernet LANs (Local Area Networks) and to IEEE 802.3 LANs, both of which use CSMA/CD (Carrier Sense Multiple Access with Collision Detection).

19.1.1 Goals

The goals of the NI function processor are as follows:

- To provide NI support for the data link layer of DECnet, Phase IV and Phase V
- To support multiple NI controllers (XNAs)
- To provide services for applications running in both kernel mode and user mode
- To provide services for multiple pieces of upper-layer software that run concurrently
- To provide hooks for diagnostics, as needed
- To utilize the NI controller to its maximum transmit/receive speed

19.1.2 Features Not Implemented

The NI function processor does not offer the following features:

- Load balancing between controllers
- Services that allow a remote node to control, dump, load, or diagnose the local node
- Shared *SAP*, *Protocol type*, or *SNAP Protocol ID*

19.1.3 Capabilities

The NI function processor performs the functions of the data link layer as specified by the DIGITAL Network Architecture (DNA). The main tasks of the NI function processor are as follows:

- Filtering the data link header of each incoming packet
- Delivering each incoming packet to its corresponding upper-layer software
- Generating the data link header of each outgoing packet
- Maintaining status information about the data link layer

The NI function processor provides the following services to the upper-layer software:

- Configuration Services—By calling these services, an upper-layer software specifies the following information:
  - What kind of packet the upper layer receives from the NI function processor (for example, Ethernet or IEEE 802.3).
  - What kind of packet the upper layer sends to the NI function processor. (That is, the NI function processor specifies the data link header information for outgoing packets).
  - How many receive buffers the NI function processor is to allocate.
- Data exchange services—Unlike the configuration services, which specify how packets are to be sent and received, the data exchange services do the actual sending and receiving. That is, these services allow an upper-layer software on one node to exchange packets with a network program on another node.

  The data exchange services, however, provide only datagram service. The NI function processor therefore guarantees neither that transmitted packets have been received nor that packets have been delivered in the order sent.
- data link status services—The data link status services provide counters and events that relate to the NI data link layer.

19.1.4 Interface with the Upper Layer

The upper-layer interface to the NI function processor is a set of procedure calls that set the physical characteristics of controllers, specify the criteria for filtering incoming packets, and transfer data. These procedure calls are of three types: request and execute I/O functions, synchronous I/O call functions, and callbacks.

19.1.4.1 Request and Execute I/O Functions

The request and execute I/O functions are invoked by calling the channel object service routines Request I/O (*e$request_io*) and Execute I/O (*e$execute_io*). These functions support the following function codes:

- *io$c_ni_ready_fpu*—Configures the NI controller, and sets the state of the FPU (function processor unit) to ONLINE. The information supplied includes the *physical address* of the controller, the CRC generation, and other related information. The configurations of the NI controller cannot be changed when the FPU state is ONLINE.
- *io$c_ni_unready_fpu*—Changes the FPU state to AVAILABLE, and resets all aspects of the controller configuration to their initial values; for example, the physical address is set to the hardware address, and the counters are set to zero.
- *io$c_get_fpu_information*—Returns configuration information about the controller. This information includes the default hardware address, the current physical address, and the maximum packet size.
- *io$c_ni_get_fpu_counters*—Returns the counters associated with the controller (for example, the number of packets sent, and the number of packets received).
- *io$c_ni_set_fpu_counters*—Sets the counters associated with the controller to zero.
- *io$c_ni_configure_channel*—Sets the network address of the channel, and sets the criteria by which the channel filters incoming packets. (In this chapter, the term *network address* refers to a value that uniquely identifies an upper-layer software, only within the local node. This value is not to be confused with any network-wide identifier.) The code *io$c_ni_configure_channel* specifies the upper layer's packet format, protocol type, SAP (Service Access Point) address, *multicast addresses*, and related information.

- *io$c_get_channel_information*—Returns packet-filtering information associated with the channel.
- *io$c_ni_send_data*—Sends a block of data. The I/O is completed after the transmit data is sent.
- *io$c_ni_receive_data*—Receives a block of data.
- *io$c_deaccess*—Deallocates the data structures and buffers associated with the channel.

19.1.4.2 Synchronous I/O Call Functions

The NI function processor supports the following synchronous I/O call functions:

- *io$c_ni_kept_buffer_limit*—specifies the maximum number of receive buffers allowed to be kept by the upper layer software.
- *io$c_ni_enable_receive_callback*—Specifies which entry point of the upper layer is to be called when the NI function processor receives a packet that satisfies the filtering criteria of the channel.
- *io$c_ni_return_buffer*—Returns a receive buffer that the upper layer has temporarily kept.

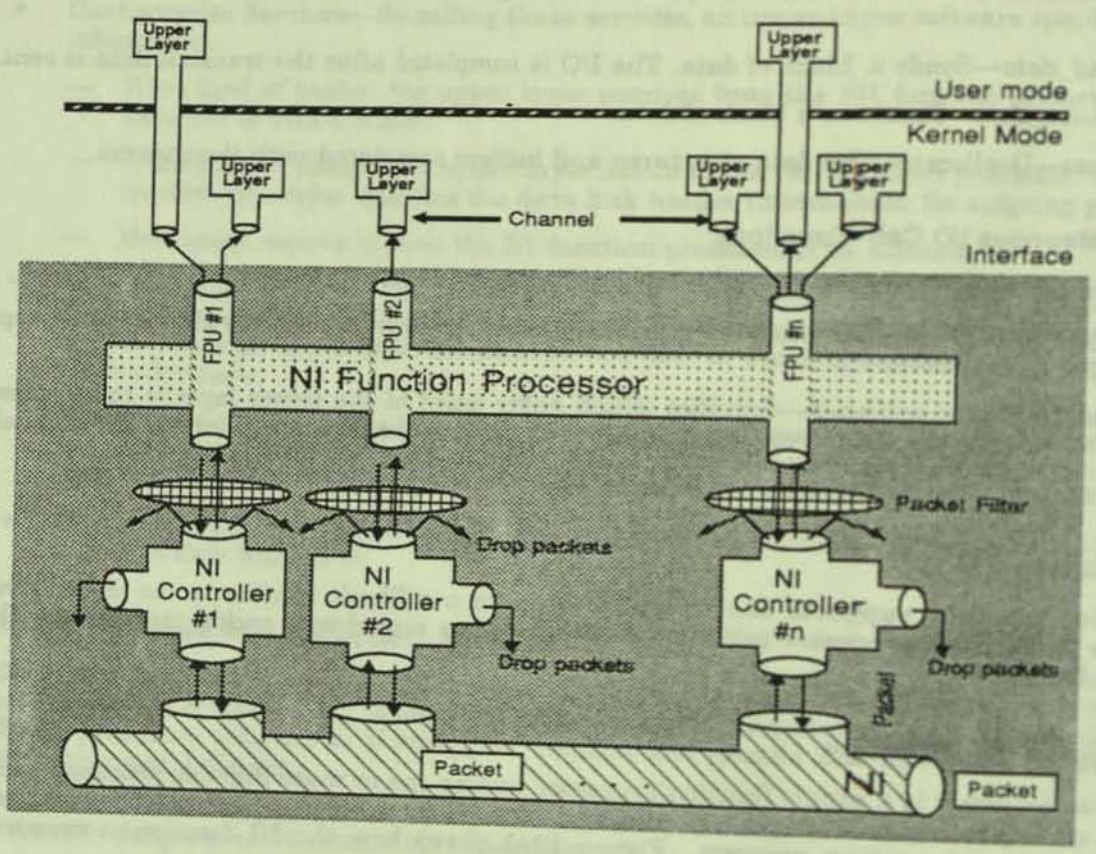
19.1.4.3 Callbacks

The NI function processor supports one procedure type as a callback routine to the upper-layer software. The NI function processor calls this procedure after receiving a packet intended for the upper-layer software.

19.1.5 Implementation Strategy

The NI function processor is a single function processor having one or more function processor units (FPUs), each corresponding to a single NI controller. The NI function processor is loaded and initialized by the autoconfiguration program. Figure 19–1 shows how the NI function processor is implemented.

Figure 19–1: How the NI Function Processor is Implemented



To communicate through a particular NI controller, an upper-layer software first creates an I/O channel to the NI FPU corresponding to that controller. The upper layer then prepares the channel for reception and transmission.

To prepare the channel for reception, the upper layer specifies the criteria by which the NI function processor selects or rejects the packets received on the channel. (For example, one such criterion might be that the upper layer receives only Ethernet packets.) Using these criteria, the NI function processor then filters packets received on the channel, delivering to the upper layer just those packets selected.

To prepare the channel for transmission, the upper layer specifies the format of the packets it sends, and its network address. With this information, the NI function processor builds the data link header of the packets that the upper layer sends.

19.1.5.1 Transmit

The transmit buffer is allocated by the upper-layer. To send a packet, the NI function processor uses the thread requesting that the packet be sent. The NI function processor provides *io$c_ni_send_data*, which is a *e$request_io* function, to transmit a packet.

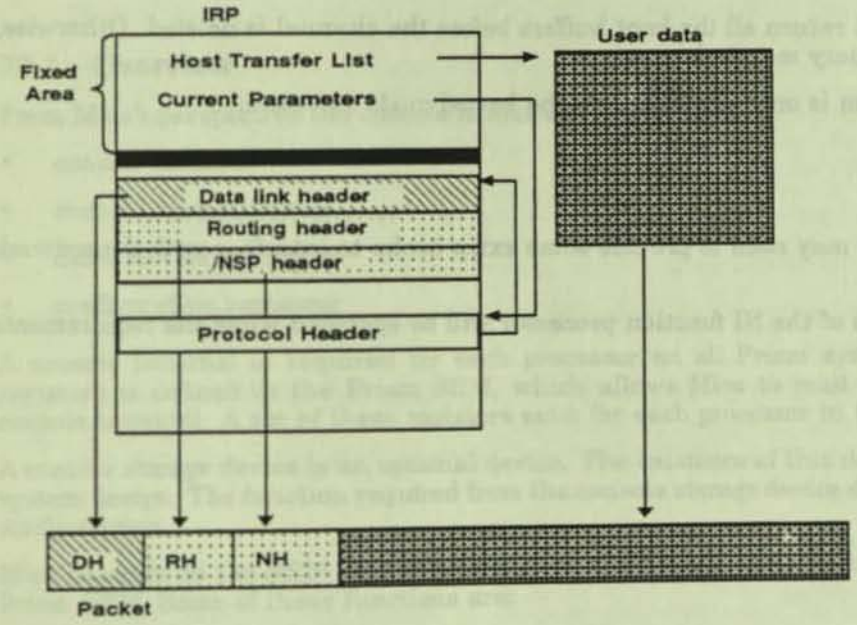
When calling the NI transmit functions, the upper layer passes two main pieces of information about the transmit buffers: the first piece is a buffer address, usually containing the upper layer's protocol header. The second piece is the address of the host transfer list located in the IRP (I/O request packet).

The upper layer, having received the size of the NI data link header from the function *io$c_get_channel_information*, leaves enough space in front of its own protocol header to build a data link header. The NI function processor first adds the data link header in front of the upper layer's protocol header. Then, using the buffer-chaining feature of the controller, the NI function processor combines the resulting header with the upper-layer data being sent to the NI controller.

For the user-mode upper layer, the NI function processor reserves the data link header buffer in the IRP when the FP parameter record is established. The NI function processor then sends the data link header with the data to the NI controller.

The I/O function is completed after the transmit buffer is sent.

Figure 19–2: Mapping of Transmit Buffers to Actual Packet



19.1.5.2 Receive

There are two mechanisms provided by the NI function processor for the upper-layer software to receive incoming data: one is the *io$c_ni_receive_data* request I/O function and the other is the *receive call back* mechanism.

Using io$c_ni_receive Request I/O Function

When an incoming packet arrives, if there is an outstanding receive posted on the channel, then the NI function processor stores the address of the NI receive packet in the IRP, and completes the I/O request. When the I/O completion routine executes, the NI function processor copies the message from the NI receive buffer into the buffer specified in the IRP and returns the NI receive buffer to the NI receive buffer pool.

If there is no IRP outstanding, the NI receive packet is queued to the channel, and the NI thread returns to wait for another message.

To Use Receive Callback

After receiving a packet intended for the upper layer, the NI function processor issues a callback to the upper layer. That is, the NI function processor directly calls the entry point that the upper layer specified in its call to the *e$synchronous_io_call* function *enable_receive_callback*.

When the upper layer receives a packet, it either immediately releases the receive buffer containing the packet, or it keeps the buffer for some period of time. If the upper layer releases the receive buffer immediately, the NI function processor returns the buffer to the receive-buffer pool as soon as the thread that delivered the packet returns. If, instead, the upper layer keeps the receive buffer for a while, the following steps occur. To release the buffer, the upper layer calls the NI function processor's Synchronous I/O Call (*e$synchronous_io_call*) function *io$c_ni_return_buffer*. Then, the NI function processor returns the receive buffer to the receive-buffer pool.

The upper-layer software must return all the kept buffers before the channel is deleted. Otherwise, the system nonpaged pool memory may be lost.

The receive callback mechanism is only supported for the kernel-mode software.

19.1.6 Outstanding Issues

1. The NI function processor may need to provide some extra hooks to interface with the network management software.
2. The diagnostic capabilities of the NI function processor will be specified when the requirements and functions are defined.

CHAPTER 20

CONSOLE SUPPORT

20.1 Overview

From Mica's perspective the console is four devices:

- console terminal
- console storage device
- SRM service processor
- configuration processor

A console terminal is required for each processor on all Prism systems. A set of four processor registers is defined in the Prism SRM, which allows Mica to read and write characters from the console terminal. A set of these registers exist for each processor in the system.

A console storage device is an optional device. The existence of this device is related to the hardware system design. The function required from the console storage device depends largely on the hardware configuration.

Mica depends on the SRM service processor of the console to perform the functions described in the Prism SRM. Some of these functions are:

- setting up the RPB
- loading EPIcode and the primary bootstrap
- setting the machine to a defined initial state
- synchronizing processors during a multi-processor bootstrap

The SRM service processor performs these functions without Mica's assistance. Mica is only a consumer of these functions, and cannot provide the console with any support for the functions.

A configuration processor is an optional device. The existence of the configuration processor is related to the hardware system design. The functions Mica performs in conjunction with the configuration processor vary greatly. Some of possible functions of the configuration processor that could be controlled by Mica are:

- reporting failed hardware
- disabling hardware modules
- executing ROM based diagnostics

20.1.1 Requirements

Mica is required to support the console terminal I/O for:

- kernel-mode debugger
- last-gasp messages
- some primitive off-line diagnostics
- bootstrap messages

Mica does *not* initially support user use of the console terminal and this includes not allowing the user to log into the system via the console terminal. On a VAX/VMS system this support is called console program I/O mode.

Mica, depending on the hardware configuration, supports the console storage device as a storage device. This device could be used for any operation in which an ordinary disk or tape could be used. Possible uses of this device are:

- system and layered-product kit distribution media
- system disk (for functions like off-line backup and initial system installation)
- storage of primary bootstrap image

The interface to this device is *not* architecturally defined, and may vary greatly between different implementations of Prism systems.

Depending on the hardware configuration, Mica works with the configuration processor. The information passed between Mica and the configuration processor is dynamic. Mica does not always reboot because the configuration has changed. Possible functions of the configuration processor are:

- notifying of failed hardware
- running low level diagnostics
- disabling hardware components
- notifying of dynamic changes to the configuration (hot swap)

The interface to this device is *not* architecturally defined and may vary great between different implementations of Prism systems.

20.1.2 Design Highlights

20.1.2.1 Console Terminal

The console terminal is a seldom-used device; however, when it is used the information transferred is usually the result of a *major* (if not catastrophic) system event.

The console terminal has two software interfaces, synchronous and asynchronous.

20.1.2.1.1 Synchronous Interface

The synchronous interface implements its functions by polling, and does not use device interrupts.

The consumers of this interface are

- kernel-mode debugger
- last-gasp messages writer
- bootstrap messages writer

This interface is required to work at an IPL higher then the console terminal's IPL.

The code is a library of routines that are linked into the image. A function-processor interface cannot be used because the code works in environments in which Mica and function processors do not exist. The interface is, however, conceptually compatible with the function-processor interface.

The code is very basic. Only simple read-and-write character functions are supported. It is critical that this code works correctly, because the system debugger uses the code.

20.1.2.1.2 Asynchronous Interface

The asynchronous interface implements its functions using device interrupts and does not use polling.

The consumers of this interface are some primitive off-line diagnostics which cannot be run throught the system management interface.

A function-processor interface is used. A full Mica system is running when this interface is used.

Only simple read-and-write character and line functions are supported. Complex terminal support, like command-line editing, is *not* supported. This is a level of functions supported by a port driver on VMS.

20.1.2.2 Console Storage Device

Mica support of the console storage device is largely dependent on the hardware configuration. The hardware configuration determines whether there is a console storage device accessible to the Mica software, and which functions the device is required to support.

The console storage device is accessible through a standard function processor interface (for example, the logical-block-unit interface). Other function processors may be layered on top of the console-storage-device function processor.

Unlike the console terminal, there is not a separate console storage device for each processor. The hardware may implement the console storage device interface as per processor registers. The console-storage-device function processor makes the console storage device appear once per system and not once per processor. The console-storage-device function processor does not require processes using the console storage device to have processor affinity.

The console-storage-device function processor also supports the concept of more than one console storage device (for example, a tape and a disk). Each device has a separate function-processor unit interface.

20.1.2.3 Configuration Processor

Mica can only support the configuration processor if it is present in the hardware. The hardware design determines what configuration-processor functions are supported, and defines the interface.

Possible Mica users of these configuration processor functions are hardware-reconfiguration software, diagnostics, error logging, and system management.

Depending on the functions supplied, both a synchronous interface and an asynchronous interface are provided.

The interface to these functions is accessible only by system software.

20.1.3 Issues

1. Rock Support

   The console terminal is well defined, and a single implementation of the support runs on all Prism processors, including Rock.

   The console device is not well defined, and a new implementation is required for each new system. It is not yet known if Rock has a console device, and thus the interface to the device is *not* defined.

   The configuration processor is not well defined, and a new implementation is required for each new system. It is not yet known if Rock has a configuration processor and thus its interface is *not* defined.

2. Multi-processor Support

   The Rock console allows the software to control the processors in the Rock system through commands defined in the SRM. The support should allow useful and flexible control of each of the processors. The method for displaying Rock's 16 logical console terminals on one physical console terminal will require careful design. Mica does *not* control the display on the physical console terminal.

CHAPTER 21

MESSAGE FUNCTION PROCESSOR

21.1 Overview

This paper summarizes the design and function of the Mica message function processor.

The message function processor passes messages from writing threads to reading threads, thereby allowing processes to communicate with one another. One use of the message function processor is as a log for the following kinds of system events: accounting, operator, security, network, and error.

Threads read and write to the message function processor through its function processor units (FPUs). Figure 21–1 shows how writing threads (Writers) and reading threads (Readers) access FPUs of the message function processor.

21.1.1 Functionality

The message function processor meets the following requirements.

Sharing

- Shared Message Streams—Reading threads sharing an FPU can also share a single *message stream*, that is, a single outgoing connection from the FPU. By default, each message in the stream is read by only one of the threads, which take turns reading.
- Shared Messages—When threads share an FPU but do not share a message stream, each thread can read all messages written to the FPU.

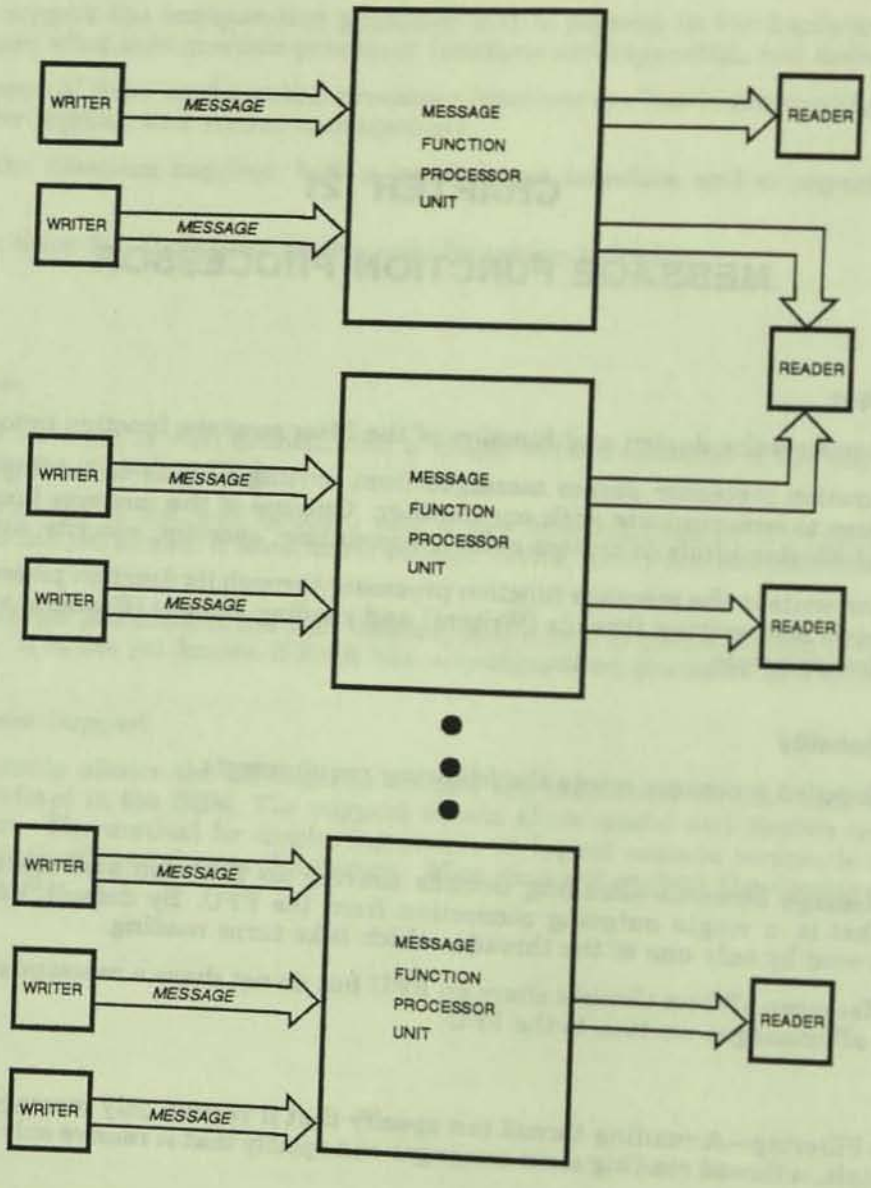
Selectivity

- Message Filtering—A reading thread can specify that it receive only messages of a certain type. For example, a thread reading error messages can specify that it receive only messages signaling disk errors.
- Selective Retention of Messages—Depending on how a message FPU is created, it *a*) retains messages if no thread is currently registered to read them, or *b*) discards such messages. For instance, an FPU receiving events (messages) during system startup retains each message until a thread registers to read it. In contrast, an FPU receiving accounting messages discards each message unless a thread is currently registered to read it.

State Control

- Enabling and Disabling Logging—If the state of a message FPU is changed from ONLINE to AVAILABLE, the FPU rejects new requests to read or write. Using this feature, the system manager can enable or disable the logging of various system events.
- Monitoring State—When state of a message FPU changes between ONLINE and AVAILABLE, the message function processor notifies all threads registered on the FPU.

Figure 21–1: How Threads Read and Write through Message FPUs



Synchronous or Asynchronous I/O

- Writing—When writing on an FPU, a thread specifies *a*) that the operation complete immediately *[write_now]*, or *b*) that it complete only when the corresponding read operation completes *[write]*.
- Reading—When reading from an FPU, a thread specifies *a*) that the operation remain incomplete until there is a message to read *[read]*, or *b*) that the operation complete immediately, succeeding only if there is currently a message to read *[read_now]*.

Other Requirements

- Resource Control—Through quotas, the message function processor limits its use of system memory.
- Kernel-Mode Support—The message function processor provides buffers so that threads can write messages in kernel mode with minimal delay.
- Performance Monitoring—The message function processor keeps an accessible set of statistical information, such as when the message function processor was most recently accessed, and how many Readers are currently registered.

21.1.2 Design

Following are the main components of the message function processor:

- Function processor units (FPUs)—The message function processor has one or more FPUs.
- Lists of unread messages—There is one list for each FPU.
- Message stream header—A thread reads on its FPU through a message stream header, of which each FPU has one or more.
- Object service routines and I/O function codes—These routines and codes operate on message FPUs. (All entries into the message function processor are through procedure-based calls; there are no system threads.)

There are two types of message FPUs: the first is always ONLINE; the second is ONLINE only if Readers and Writers both are registered on it. The first type buffers unread messages that lack Readers; the second type discards such messages.

When a thread creates an FPU, the system charges the requesting thread for the pool necessary to create the data structures of the FPU, and for the pool to buffer messages written to the FPU. Similarly, whenever a thread performs a *[write]* operation on an FPU, the system also charges the writing thread for the pool to buffer the message. (In contrast, *[write_now]* entails no such charge.)

Before accessing the message function processor for the first time, a thread must register on a message FPU as a Writer, a Reader, or both. When a reading thread tries to register on an FPU through a nonexistent message stream header, the message function processor creates a new message stream header. A reading thread can, however, register through an existing message stream header. The registering thread then shares the message stream header (and thus the message stream) with Readers already registered on that block.

A writing thread assigns a 64-bit *message type* to each message written. The message type shows the kind of message written. When registering, each reading thread specifies the message types it accepts. After registering, a reading thread can change this specification, and thereby accept different message types.

21.1.3 Functional Interface

The message function processor contains an entry for the following I/O executive service calls:

- *execute_io*
- *synchronous_io_call*
- *initialize_io_parameters*
- *create_fpu*
- *delete_fpu*
- *cancel_io*

- *get_fpu_information*

The message function processor implements the following Request I/O function codes:

- *register*—Registers a thread as a Writer, a Reader, or both. Specifies the message types that a thread reads.
- *read*—Completes only after the attempt to read either succeeds or fails.
- *read_now*—Completes immediately.
- *write*—Completes only after all Readers have read the message.
- *write_now*—Completes immediately.
- *deaccess*—Called by the I/O system after all threads have closed their channels to the message function processor. Deallocates data structures and does related cleanup.
- *change_types*—Changes the messages types that a thread reads.
- *ready_fpu*—Changes the state of an FPU from AVAILABLE to ONLINE.
- *unready_fpu*—Changes the state of an FPU from ONLINE to AVAILABLE.
- *enable_fpu_state_change_ast*—Requests that a thread receive an AST each time the state of its FPU changes.
- *disable_ast*—Disables delivery of state-change ASTs.

The message function processor has two entry points through *synchronous_io_call* used only by threads writing in kernel mode:

- *allocate_buffer*—Gets a buffer.
- *queue_buffer*—Queues a buffer.

CHAPTER 22

PRISM DIAGNOSTIC MONITOR

22.1 Overview

22.1.1 Introduction

The PRISM Diagnostic Monitor (PDM) is the controlling environment for all loadable PRISM diagnostic programs that execute in the on-line or off-line run-time environments (described in a following paragraph). Its purpose is to provide the following functions:

- A user-diagnostic interface, which ensures that all diagnostic programs present a consistent and convenient user interface.
- A set of diagnostic services and other routines that facilitate the writing of diagnostic programs.
- A mechanism for controlling and monitoring the execution of diagnostic programs, singularly or in parallel.

22.1.2 Diagnostic Run-time Environments

For PRISM systems, several diagnostic run-time environments are used. These environments are:

- Self-test

  The self-test environment requires no loadable software. Tests may be ROM resident, or they may be built into the hardware as "built-in self tests" (BISTs). These tests are activated by a console command.

- Standalone

  Diagnostic programs in the standalone environment may be loadable or ROM resident. They are free standing in that there is no higher-level software controlling them.

- Off-line

  Diagnostics in the off-line environment are not free standing. They require the use of system-type software for I/O and other purposes. For PRISM, the off-line environment is defined as a subset of MICA, with all of the functionality of MICA except:

  — It does not employ paging out.

  — It does not require a client. (Testing of links to clients is possible, however.)

  — It can be booted from a storage device located on a console service processor, if such storage device exists.

  — The only user interface is through a local internal or external service processor terminal.

  — It can be used only by specific applications such as diagnostics, installations, or off-line backups.

The minimum hardware needed for running the PRISM off-line environment are one functioning scalar processor, a TBD-sized amount of memory, a boot device (located either on the console or on one of PRISM's I/O buses), and a console terminal.

- On-line

  The on-line environment is the full operating system, with or without users. It is generally the case that other users are present while diagnostics are being executed. Depending on the type of testing taking place and the type of device being tested, a device under test may or may not have to be exclusively allocated to the diagnostic program performing the testing. As is the case in the off-line environment, diagnostics are not free standing and require the use of system services for I/O and other purposes.

For PRISM, self tests are sometimes executed in the on-line environment. That is, a particular subsystem (such as a single processor) is taken off line by MICA for testing purposes, but the operating system is still executing and users are still present.

The rest of this discussion deals only with the off-line and on-line diagnostic environments.

22.1.3 Functional Overview

PDM is used in both the off-line and on-line environments. It provides diagnostic users and diagnostic programs with consistent interfaces in both environments. Thus, the same diagnostics are used in both environments, with the same user interface.

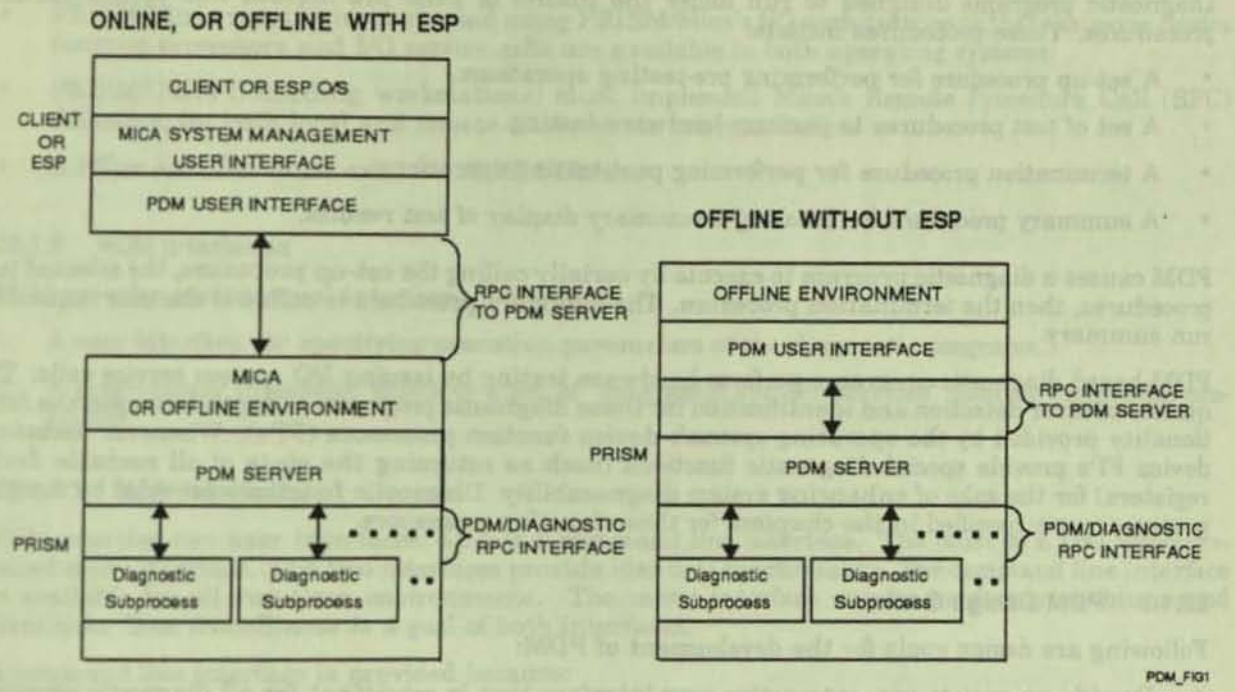
PDM is implemented in two images, running as separate processes. One image contains the user interface and the other, referred to as the "PDM server", contains all other functionality. The user interface is implemented separately because in some run-time environments it resides in a separate system. Specifically, in client-server environments, such as Glacier and Cheyenne, the user interface resides in a client system, while the PDM server, along with the diagnostic programs, resides in PRISM. Off line, the user interface resides in the external service processor, if one exists. Otherwise the user interface resides in PRISM, along with the PDM server. There is a separate user interface process and server process for each user.

The user interface and PDM server communicate via a Remote Procedure Call (RPC) interface. It is immaterial whether the communicating elements are local to one another or remote; the RPC interface makes communication path differences transparent to PDM.

Each diagnostic program is a separate image file, executed as an individual subprocess of the PDM server. There is a separate diagnostic subprocess for each device selected for testing. This is true even if two identical devices are being tested by the same diagnostic program image. These subprocesses can be run in parallel or they can be activated serially, depending on the choice of the operator. The PDM server communicates with the diagnostic subprocesses via an RPC interface.

Figure 1 illustrates the layout of PDM and diagnostic subprocesses, for both on-line and off-line environments.

Figure 22–1: Layout of PDM and Diagnostic Subprocesses



The sequence of operation is as follows:

1. To start the PDM user interface, a user issues a command at a terminal. (If the Mica system management user interface is present, PDM is accessed from it.) This causes the user interface process to be activated, which in turn starts the PDM server process.
2. The user, via the PDM user interface, selects the devices to be tested, the types of tests to be executed, and other run-time options.
3. The user issues a "start testing" command.
4. The PDM server creates a subprocess for each selected device, passing run-time parameters to the subprocess.
5. The subprocesses execute. If they need to report errors or other information to the user, they do so by sending messages to the PDM server. The user is provided with a dynamic display of current execution status.
6. When testing is complete, PDM notifies the user. Diagnostic subprocesses are not killed at this point, so that the user can obtain a run summary or restart the same tests.
7. The user may restart the same set of tests, or change the selected set. This is effectively "go to step two". Before testing starts, all diagnostic subprocesses created during the previous run are deleted.
8. When finished testing, the user exits the PDM user interface, causing the PDM server process and all diagnostic program subprocesses to be deleted.

22.1.4 Components of a PDM-based Diagnostic Program

Diagnostic programs designed to run under the control of PDM are divided into several callable procedures. These procedures include:

- A set-up procedure for performing pre-testing operations.
- A set of test procedures to perform hardware testing.
- A termination procedure for performing post-testing operations.
- A summary procedure for creating a summary display of test results.

PDM causes a diagnostic program to execute by serially calling the set-up procedure, the selected test procedures, then the termination procedure. The summary procedure is called if the user requests a run summary.

PDM-based diagnostic programs perform hardware testing by issuing I/O system service calls. The quality of error detection and identification for these diagnostic programs is dependent upon the functionality provided by the operating system's device function processors (FPs). Whenever necessary, device FPs provide special diagnostic functions (such as returning the state of all readable device registers) for the sake of enhancing system diagnosability. Diagnostic functions provided by function processors are specified in the chapters for those function processors.

22.1.5 PDM Design Goals

Following are design goals for the development of PDM:

- Provide an easy-to-use, interactive user interface that is consistent for all diagnostic programs and identical in all run-time environments (on-line Mica, on-line Ultrix, off line).
- Provide a design that is easily extendible to all future PRISM implementations.
- Provide a design that can support both Mica and Ultrix on-line environments with minimal difficulty.
- Assist in providing diagnostic programmers with a method to easily write a set of diagnostic programs that have a consistent user interface, that are portable between different operating systems (Mica and Ultrix), and that will function without modification on future PRISM implementations.
- Ensure that the design of PDM does not inherently place limits on such diagnostic test characteristics as level of error detection or isolation. (Specific error detection and isolation goals are spelled out in diagnostic project plans for PRISM products.)

22.1.6 PDM Design Non-goals

Following are non-goals for PDM development:

- Provide on-line diagnostic support for a ported PRISM/Ultrix that does not meet the requirements listed below.
- Provide a diagnostic environment compatible with existing VAX diagnostic products.

22.1.7 Requirements on Other Products for Meeting Design Goals

- PRISM/Ultrix must be implemented using PRISM/Mica's I/O architecture so that the same device function processors and I/O service calls are available in both operating systems.
- PRISM/Ultrix (including workstations) must implement Mica's Remote Procedure Call (RPC) definition, for both local and remote interprocess communication.
- A Pillar compiler must exist for PRISM/Ultrix.

22.1.8 PDM Interfaces

PDM provides two external interfaces. These are:

1. A user interface for specifying execution parameters of the diagnostic programs.
2. A programmer interface, which is a set of "diagnostic system services" used by diagnostic programs.

22.1.8.1 User Interface

PDM provides two user interfaces. One is a command line interface. The other is a DECwindows-based menu interface. The two interfaces provide identical functionality. The command line interface is available for all run-time environments. The menu interface can be used only on bitmapped terminals. User friendliness is a goal of both interfaces.

A command line interface is provided because:

- It allows "scripts" or command files of PDM commands to be used.
- It works on any type of terminal.
- Some users prefer command line interfaces. Experienced users can issue commands more quickly with the command line interface than with the menu interface.

DECwindows is used for implementation of the menu interface because:

- It is possible to produce menus in a format consistent with system management's user interface, which also uses DECwindows.
- DECwindows provides a convenient means for producing well-designed, highly-interactive windows.
- DECwindows and bitmapped terminals are considered "state-of-the-art" and make use of DEC's newer software and hardware products. Using these products is good salesmanship.

In the on-line environment, the PDM user may be located at a terminal on a client system, or may use the terminal connected to a PRISM external service processor, if one exists. Off line, the user is located at the PRISM's internal or external service processor terminal (whichever one exists).

The user interface allows a diagnostic program user to:

- Select a set of one or more devices for testing.
- Choose which tests to run on the selected devices.
- Specify whether the devices are to be tested sequentially or in parallel.
- Individually start and stop the selected tests.
- View a run summary during testing or after testing has completed.
- Obtain Help information for any PDM user selection, or for the diagnostic tests.
- Run a command file of PDM commands.

- Switch back and forth between menu mode and command line mode, if the environment supports both modes.

22.1.8.2 Programmer Interface—Diagnostic Services

The PDM programmer interface is a set of "diagnostic services" plus pre-defined Pillar in-line procedures and type definitions which facilitate the creation of PDM-based diagnostic programs.

Diagnostic services are calls from the diagnostic to the PDM server. They provide the interface between a diagnostic program and the user's terminal. There are calls for reading from and writing to the user's terminal. These services also provide such capabilities as allocating and deallocating devices, fetching the name of the device under test, or writing error log records.

Pre-defined in-line procedures and/or type definitions are used to define the various callable diagnostic procedures (set-up, test, etc.) and local data structures required for a PDM-based diagnostic.

22.1.9 PDM Internal Interfaces

22.1.9.1 Interface Between the User Interface and the PDM Server

The interface between the user interface and the PDM server is based on RPCs. Whether PDM is used in a client-server configuration or it is entirely PRISM resident, the calls are identical. User input is fetched from the user terminal by the user interface and passed as an argument to a call to the PDM server. Likewise, output is passed as an argument from the server to the user interface.

22.1.9.2 PDM/Diagnostic Interface

The PDM/Diagnostic interface is the means by which the PDM server controls the execution of diagnostic program subprocesses. All communication between these processes is by means of an RPC interface. This RPC interface is separate from the user I/O RPC interface. For this interface, the PDM server is actually the client, and the diagnostic processes are considered to be multiple servers. The PDM server acts as a client to access the diagnostic's callable procedures (e.g., set-up, tests, etc.). Calls in the opposite direction, from the diagnostic programs to the PDM server, are also used. Calls in this direction PDM server are used for requests of user terminal I/O (to fetch user input or display error status) or to send a "testing in progress" indication.

22.1.10 PDM's Interfaces to the On-line and Off-line Environments

In the on-line run-time environments (for FRS, Glacier and Cheyenne), users access PDM via Mica system management's user interface. Off line, PDM is accessed through the off-line user interface.

In all run-time environments, terminal I/O for implementation of the command line interface is accomplished via TBD calls to the runtime environment. DECwindows is used to implement the menu interface.

22.1.11 Other PDM Features

- Diagnostic QA

  PDM provides a rudimentary quality assurance feature for diagnostic programs. This feature provides simple validation of the operation of diagnostic tests by performing such functions as executing tests for multiple passes, executing tests in random order, and other TBD operations.

- Installation

  Installation of PDM image files involves three separate issues. These are:

  - Installation of PDM image files into Mica. These files include the PDM server image and associated files, such as message section files, diagnostic image files, and the PDM user-interface image for off-line, non-ESP operation.

- Installation of a VAX/VMS-formatted PDM user interface image onto a VAX/VMS-based client or ESP.
- Installation of diagnostic image files into the PDM's diagnostic database.

Specific installation procedures are TBD.

- Message section files

  All ASCII text for PDM and for diagnostic programs is stored in message section files, to allow multiple language support. Language selection is performed on a per-user basis. All message section files are stored on PRISM. Mica's message utility will be used for creating, storing, and referencing all message text.

22.1.12 Security Issues

On line, when running PDM from a client system, access to PDM is controlled by Mica's security features. Refer to the chapter describing system security. When running PDM from PRISM's external service processor, the ESP's VAX/VMS login and user privileges control access to PDM.

Off line, when running from PRISM's ESP, the ESP's VAX/VMS login and user privileges control access to PDM. When the user's terminal is the internal service processor's terminal, any security features that exist are provided by the console service software. These features are TBD.

22.1.13 Changes from Rev. 1.0 of "The PRISM Diagnostic Environment"

- Support for a Diagnostics/Utilities Protocol (DUP) programming interface is not being provided. Instead, a "DUP Dialogue Driver" is being supplied. This is a program that runs under PDM control and provides a user interface to diagnostic programs that run locally in MSCP-type controllers. These diagnostics use DUP to communicate with the host processor. The DUP dialogue driver allows a user to select which diagnostics to run, and it receives and displays error or other messages from the diagnostics.

CHAPTER 23

ERROR LOGGING

23.1 Overview

23.1.1 What Is Error Logging?

Error logging is the creation of a permanent, referencible record of system hardware and software errors, along with other relevant events. This record is kept online and can be monitored either dynamically, as events take place or in retrospect, after event information is stored, to determine when error rates begin to reach a dangerous threshold. The log can also be used for analysis of what errors or events took place and when they occurred. The goal of such analysis is to identify patterns that may indicate potential catastrophic failures before they occur, so that corrective action may be taken before system downtime results, thus providing greater system availability and reliability.

23.1.2 How Is Error Information Stored?

For PRISM, errors and events are recorded in an error log file. This file is sequential and resides, by default, on the Mica system disk. The file is called the "system error log file", to differentiate it from other possible error log files, such as the "off-line error log file" (see Section 23.1.13).

23.1.3 What Does the System Error Log File Contain?

The system error log file consists of error log "records". Each record represents one detectable error or event, and contains all of the recorded information about that error or event. "Errors" include all software-detectable hardware or software errors. "Events" consist of those system events that are worth noting because they are useful in determining the cause of subsequent failures. Such events include media mounts, dismounts, and volume changes; hardware configuration changes; operating system parameter changes; system boots, reboots, and recoveries from power failures; and the initiation and completion of diagnostic programs.

Additionally, the log file contains a copy of all system configuration information available to Mica. System configuration information includes the types and number of devices, and can include such details as module serial numbers. Whenever a new error log file is created or the system is rebooted, a copy of the current system configuration information (obtained from the system configuration file) is placed at the beginning of the file. When configuration changes occur, the changes are logged as events. Thus by scanning the log file, the full configuration for a given point in time can be determined.

23.1.4 What Do System Error Log Records Contain?

Each system error log record contains all of the relevant information about the error or event being reported that is available to the software reporting the error. Such information might include device register contents, MSCP error report contents, or machine check frame pointers. The types of records supported, along with the exact contents of each record, are provided in the full error logging chapter. Additionally, a time stamp and a sequential event number are written in each record.

Sometimes a single error results in multiple system events, such as multiple MSCP messages or the case of an error being detected and reported at both a low level (e.g., a device FP) and a higher level (e.g., a file system error). In such cases the system attempts to assign one event number to multiple records. (Each IRP created is assigned a sequence number, and device FPs are provided with a procedure which returns the sequence number and other record header information.)

23.1.5 Who Creates System Error Log Records?

System error log records are created by Mica software whenever a recordable error or event is detected. The general rule is that the process or thread detecting the error or event is responsible for creating an appropriate error log record. Thus, a device hardware error detected by a device function processor is reported by that function processor. Or, the record for a machine check exception is created by the condition handler. The process or thread creating the record provides all of the error-specific or event-specific record fields.

23.1.6 How are Records Placed into the Error Log File?

Once a record has been created, it is simply passed as a message to the error message FPU. A separate process, called the "opcom server", is responsible for reading error log record messages. It does this by creating a reader thread for the error message FPU, called the "error message reader". This thread reads records from the message function processor and saves them in a buffer. (Refer to the chapter on operator communications for a discussion of message FPU reader threads.)

Once a TBD number of records have been collected or a TBD amount of time has passed, the error message reader calls the appropriate I/O service and the records are appended to the system error log file. If the system error log file doesn't exist (e.g., the system manager has removed or renamed the file), a new one is created.

23.1.7 How Can the System Error Log File be Read?

Error log entries are stored as binary records. To create readable displays of these records, an error record formatting utility (ERF) is provided. This utility is able to recognize each type of error log record and provide a display that can label each field within the display. ERF also has the capability of finding and displaying subsets of error log records, such as all errors for a specified device, all errors within a time frame, or all errors of a specified type. ERF output is be displayed on a terminal, or it may optionally be written to a file.

23.1.8 Where Does ERF Reside and Execute?

ERF resides and executes on the PRISM system. It runs under the PRISM Diagnostic Monitor (PDM). PDM allows ERF to be environment-independent. Thus ERF can run on Glacier, Cheyenne, Ultrix-based PRISMs, or off line without modification. (See Section 23.1.13 for a discussion of off-line error logging.)

23.1.9 Where Do ERF Users Reside?

Since ERF runs under PDM, the PDM user environment controls the location of the ERF user terminal. Thus, when the PRISM is online in a client-server configuration, the user terminal can be a client terminal or PRISM's VAX/VMS-based external service processor, if one exists. Off line, the user sits at PRISM's internal or external service processor terminal (whichever one exists).

It is also possible to run ERF from a remote CSC diagnostic site. When the PRISM/Mica system is online, the remote connection is made through a client. When the PRISM/Mica is off line, remote access is made through the PRISM console.

23.1.10 Who can Access the System Error Log File?

Special access rights are required for accessing the system error log file online. Access is controlled by the fact that a user must enter the system management software in order to reach ERF. (The user runs system management, selects the diagnostic environment [PDM], then selects ERF.)

23.1.11 Who can Control Error Logging?

Access rights are required for controlling system error logging. The only operator-controllable error logging options are starting and stopping the error message reader, and deleting or renaming the system error log file. Starting and stopping the error message reader can be accomplished only via the system management interface. Deleting or renaming the error log file is controlled by file ownership and protection codes.

23.1.12 How Is the Error Log Data Used?

There are three ways in which error log data can potentially be used.

- Via ERF, system managers or field service can view error log contents to determine where and when errors are occurring. System reconfigurations or other corrective actions can be taken, based on this viewing of the log contents.
- A process running under Mica can monitor the error records as they are being produced and notify the system manager and/or a field service office if error occurrences approach a predefined threshold. This or another process could also provide a dynamic display of system error activity.
- An intelligent, rule-based program can analyze the error records. It can be used in an attempt to predict future hardware failures based on error and event records being written to the log.

23.1.13 How Does Off-line Error Logging Work?

It is possible to enable the error message reader under the off-line environment. This is accomplished via an off-line-only option under PDM. When enabling off-line error logging, the user must specify the location and filename for the error log file. It is thus possible to write off-line records to the system error log file, or to create a separate file for off-line records. Off-line error log records are identical in format and content to online records, with the exception that the record contains a flag indicating the system is in off-line mode.

23.1.14 Are There Other Error Logging Facilities?

The error message reader, along with the log file produced by it, is the only error logging facility considered to be a part of Mica. It may be the case that some implementations of PRISM/Mica may provide supplementary error logging facilities, such as a log on the external service processor (ESP) for recording console-detectable events (e.g., crashes or EMM-readable conditions). Such a log is not considered to be a part of Mica, *per se*. However, if the ESP can send these events to Mica's error message reader, they are recorded in the system error log file and thus become a part of the system's error history.

File System

This set of chapters describes the file system components of Mica.

CHAPTER 24

DISK FILE SYSTEM FUNCTION PROCESSORS

24.1 Overview

This chapter describes the characteristics and interfaces of a particular class of function processors, referred to as the disk file system function processors, or DFFPs. The DFFP used to implement Mica's first file system for locally attached disk storage is called the Files–11 ODS2 function processor, which is the subject of Chapter 25, Files–11 ODS2 Function Processor. The first remote disk file system to implement this disk file class interface is the distributed file system client function processor, described in Chapter 46, Distributed File Service Client Function Processor.

This chapter describes the functions and I/O parameter records that a function processor must implement to conform to the disk file system class interface. This chapter does not discuss the functions used to implement operations that are specific to a function processor, such as those that initialize and ready a function processor and its function processor units (FPUs). This chapter also does not discuss internal design details.

The Files–11 ODS2 function processor accesses locally attached disk storage via a striping, shadowing, or MSCP function processor. The DFS Client function processor uses the request/response function processor to access remote file systems. Since both the Files–11 ODS2 and DFS Client function processors implement the disk file system class interface, the Mica Record Management Services (RMS) can use this interface to access any disk file, regardless of whether the file resides on locally attached disk storage, or is accessed through a remote file system.

This scheme is illustrated in Figure 24–1.

24.1.1 Files and Directories

A *file* is a named collection of data that is organized into 512-byte *blocks*. These blocks are referenced by a *virtual block number (VBN)*. A file also has a set of named *file attributes* (such as maximum record size, creation date, disk alignment, and so on), which the file system maintains separately from the file's data.

A file is named by a character string. Since there may be more than one *version* of a file with the same character string name, both the character string and version number define a *filename*.

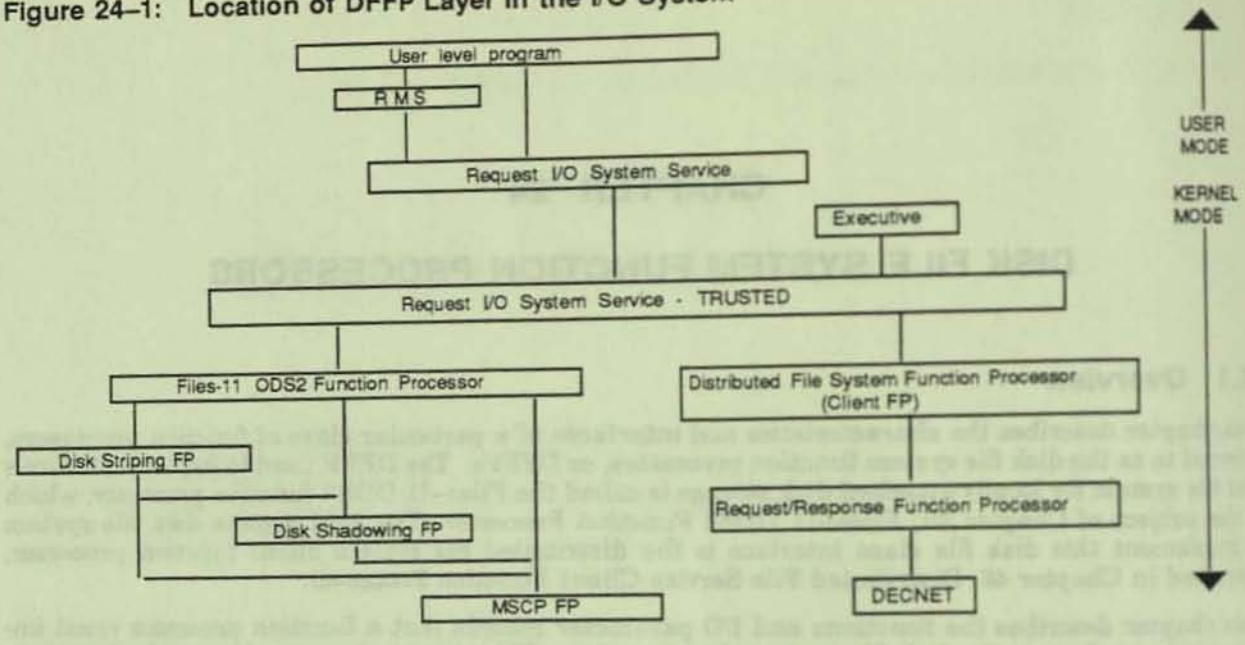
Filenames are organized into *directories*. A *directory name* is a character string that represents a directory. A filename in a directory is referred to as a *directory entry*. Directories and files exist within a given *volume*. Every volume has an implicit *root directory*.

A *directory path* is a list of directory names. The first element in the list is a subdirectory of the root directory. The second element in the list is a subdirectory of the first directory, and so on.

A *filename path* is the combination of a filename and a directory path. The final element in the directory path is the directory in which the filename is entered.

A given file may be entered into more than one directory, using one or more filenames. These are known as *synonym filename paths*.

Figure 24–1: Location of DFFP Layer in the I/O System



Each file has a *backlink* to a directory. Each directory has a backlink to its parent directory (except the root directory, of course). The filename path, as represented by the sequence of backlinks from a given file, is referred to as the file's *backlink path name*.

24.1.2 Volume Sets

In a disk file system, the largest logical unit of a disk structure is a volume, or a collection of volumes, known as a *volume set*. When volumes are organized into volume sets, the entire volume set is treated as a single volume. For example, file I/O operations, such as reading and writing, are done on the volume set, rather than on the individual volumes of the set. The implementation of volume sets is optional, and is discussed here because the Files–11 ODS2 function processor (Chapter 25, Files–11 ODS2 Function Processor) includes volume sets.

Since the interface discussed in this chapter deals with volumes and volume sets tranparently, the term *volume* is used to refer to both volumes and volume sets.

24.1.3 Objects Used By Disk File System Function Processors

Disk file system function processors deal with function processor unit (FPU) objects and channel objects. The general use and definition of these objects is explained in Chapter 8, I/O Architecture.

24.1.3.1 Function Processor Unit Objects

A DFFP represents a single disk volume or a volume set with a single FPU, known as a *volume FPU*. The details of creating and deleting a volume FPU vary with the specific implementation of the function processor. These details are discussed in Chapter 28, File Management Utilities, Chapter 25, Files–11 ODS2 Function Processor, and Chapter 46, Distributed File Service Client Function Processor.

24.1.3.2 Channel Objects

A channel object is used to perform disk file system class interface functions on a volume FPU. A channel uses the standard *io$c_fpu_access* function described in Chapter 8, I/O Architecture to access a volume FPU. Such a channel is referred to as a *volume channel*.

Once a file has been accessed on a volume channel, that channel becomes known as a *file channel*.

24.1.4 Other I/O Architecture Support

The DFFP must provide a function processor definition (FPD) object, which defines the entry points to the procedures within the function processor. The function-processor-specific support for this object is discussed in Chapter 25, Files–11 ODS2 Function Processor and Chapter 46, Distributed File Service Client Function Processor.

24.1.5 Disk File System Class Interface Functions

Following is the list of function codes that are required for a DFFP to conform to the disk file system class interface.

1. File Access and File Creation:
   - *io$c_dfile_access_file*– open a file (transform a volume channel into a file channel)
   - *io$c_dfile_deaccess_file*– close a file
   - *io$c_dfile_create_file*– create a new disk file and enter it into a directory
2. Data Transfer:
   - *io$c_dfile_read_file_data*– read virtual blocks from the file into memory
   - *io$c_dfile_write_file_data*– write virtual blocks from memory
   - *io$c_dfile_security_erase*– write a security erase pattern to virtual blocks in the file
3. Directory Entry Search and Modification:
   - *io$c_dfile_read_directory*– read directory entries from a given directory to provide for name wildcarding by services using the DFFP
   - *io$c_dfile_modify_dir_entries*– make a new directory entry, remove a directory entry and delete the file
4. Read and Write File Attributes:
   - *io$c_dfile_read_attributes*– read file attributes, directory backlink path
   - *io$c_dfile_write_attributes*– write file attributes
5. File Storage Management:
   - *io$c_dfile_allocate_storage*– allocate free storage on the volume to a file
   - *io$c_dfile_deallocate_storage*– free and return blocks to the volume pool of available blocks
6. Memory Management Support:
   - *io$c_dfile_mmclone_access*– clone file access types to another channel (for use by memory management)
   - *io$c_dfile_page_read*– read pages into physical memory (for use by memory management)
   - *io$c_dfile_page_write*– write pages from physical memory

7. Volume and Channel Query Specified by I/O Architecture:
   - *io$c_get_fpu_information*– must return the value *io$c_dfile_interface_class* for the standard item code *io$c_item_interface_class*
   - *io$c_get_channel_information*– must support the item *io$c_item_dfile_quota_info* to return information about disk usage and diskquota information.

24.1.6 Other Topics

The following topics will be addressed in more detail in the detailed design chapters:

- Failure handling

  The DFFP may encounter unexpected errors during the execution of I/O requests issued to the FPUs layered below it. It must provide a method of handling these errors, which will probably consist of passing the error code back up to the level from which the request was invoked.

  A set of standard status codes are defined by this interface.

- Caching

  Caching is optional in that it is not required to make the function processor work. However, caching may be required for acceptable performance levels. Specifically, this interface effectively requires a directory name cache similar to that implemented by RMS in the VMS system. Directory name caching is necessary because files with full directory path names are specified.

- Maintenance of disk integrity

  Utilities are provided to verify and maintain disk file system structure for Files–11 ODS2 volumes. These utilities are discussed in detail in Chapter 28, File Management Utilities.

- Access control

  Files and volumes must be protected from access by unauthorized users. This protection is provided by access control lists (ACLs). When a channel is created to a volume or a file, the ACL is checked against the identifiers of the user invoking the request. Access is denied to those users without proper access privileges. Security is described in detail in Chapter 10, Security and Privileges.

- Mount verification

  If a mounted volume goes off line for any reason, it needs to be checked when it comes back on line. The DFFP can request to receive an AST when the volume comes on line, triggering the mount verification process. Mount verification is the process of ensuring that the correct volume is still mounted; if not, it should return an error status. No operations are allowed on a volume while it is undergoing mount verification. This topic is addressed in detail in the specific function processor design chapters.

CHAPTER 25

FILES–11 ODS2 FUNCTION PROCESSOR

25.1 Overview

This chapter describes a disk file system function processor that supports Files–11 On Disk Structure Level 2, Version 3 volumes (ODS2-3), referred to hereafter as F11FP. The F11FP conforms to the interface described in Chapter 24, Disk File System Function Processors, except that the *io$c_add_to_volume_set* function is not implemented.

This chapter presents the internal operations and algorithms used by the F11FP disk file function processor interface for ODS2-3 volumes. This section summarizes the data structures and some of the methods used to implement the F11FP. The relationships between the various data structures is shown in Figure 25–1.

25.1.1 Files–11 ODS2-3 Data Structures

This section describes both Files–11-specific data structures and common disk file function processor data structures that have Files–11-specific fields. These data structures are:

- Function processor unit object

  There is one FPU object for every virtual volume; it is created when the volume is mounted. The FPU object contains:

  — List head of the file object cache

  — References to channel objects for system files

  — Pointer to the file object container for this volume

  — Primary volume control block

- Volume control block (VCB)

  There is one VCB for each volume in the volume set. The VCB contains:

  — Volume status information

  — Volume parameters and default values

  — Data from the index and bitmap files

- Relative volume table (RVT)

  There is one RVT for each volume set. The RVT contains a pointer to each VCB in the volume set.

- Volume set continuation

  A volume set continuation consists of a RVT and one or more VCBs, depending upon the number of volumes in the volume set.

- File object

The ACL on the file object is a copy of the ACL from the file header

The file object contains:

— A copy of the file header data, such as:

  * Structure level and version
  * File ID and sequence number
  * File ID of next extension header
  * File ID of directory which contains primary entry for this file (backlink)
  * File attributes
  * File access privilege levels
  * File protection code
  * Highwater mark
  * Security classification mask

— Status: marked for delete, or open for shared/exclusive read/write

— Number of channels to this file

— Access matrix

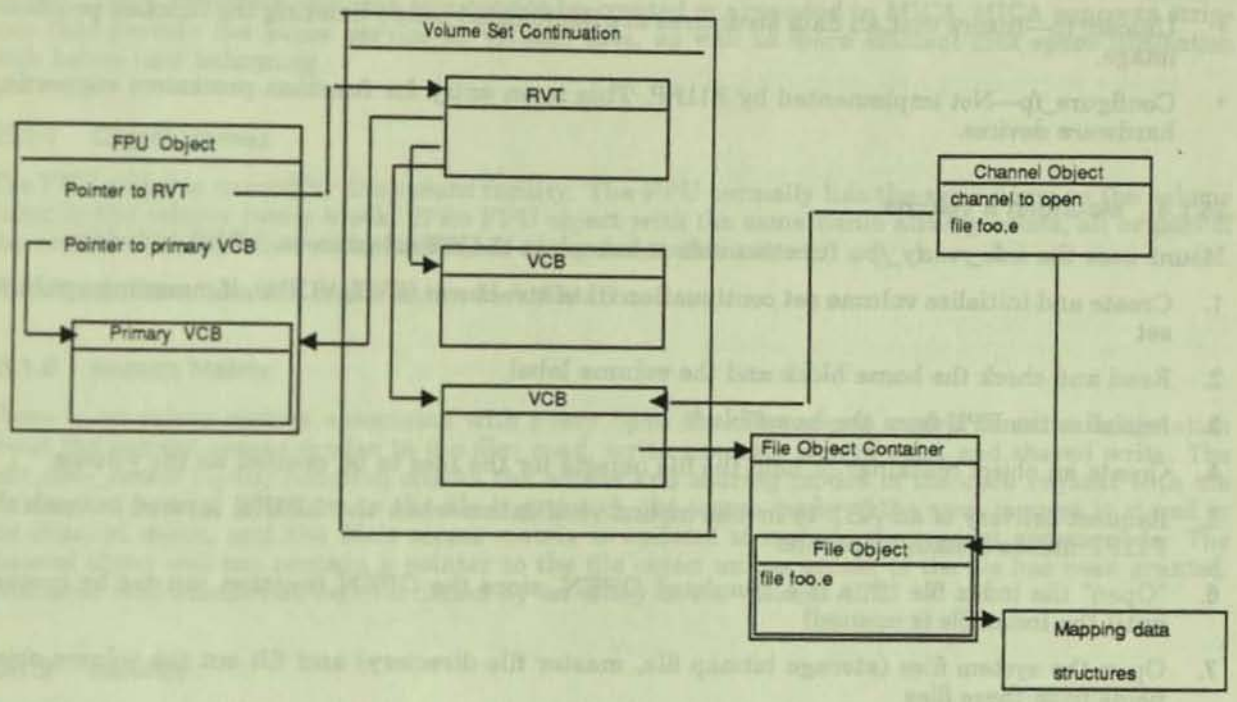
— Reference to the file's map pointer block (MPB)

— Data security erase pattern

- Map pointer block (MPB)

The map pointer block contains retrieval pointers derived from the file header map area. Each map pointer entry describes one logical extent of the file. The file object contains a pointer to the file's primary MPB. The MPB is designed to provide very fast mapping of file virtual blocks to volume logical blocks.

Figure 25–1: Relationship of Files–11 Data Structures



25.1.2 Threads

The F11FP may complete a request in the context of the calling thread by passing the request to the logical block unit (LBU) function processor described in Chapter 15, Direct Access Mass Storage Function Processors, or it may use system threads for further processing. Control operations, such as access and create, are passed off to a system thread. Reads and writes are passed on to the next lower layer, if they can be done with a single logical I/O operation. In order to free the user's thread, a system thread is used to perform read/write operations that require more than one logical I/O to complete the request. During mount verification, all reads and writes are queued to a queue serviced by one or more system threads, since the function processor is stalled until it returns to the ONLINE state.

The processing of a request may result in a recursive entry into the F11FP itself, or it may invoke logical I/O to the LBU layered beneath the F11FP. The F11FP can call itself to perform I/O functions as needed to complete a request. The create function, for example, calls the F11FP to read the index file. Recursive calls to the F11FP use the same system thread that is processing the request.

25.1.3 FPU procedures

The following procedures are called by direct entry to the F11FP via the FPD:

- Initialize_fpu—Initialize a newly created FPU object.
- Initialize_io_parameters—Allocate and initialize an IRP for a new *e$execute_io* request.
- Execute_io—Perform an I/O function (functions are listed in Chapter 24, Disk File System Function Processors).
- Synchronous_io_call—Not supported.
- Get_fpu_information—Retrieve information related to an FPU object.

- Delete_fpu—Finish dismounting the volume represented by the FPU. Cause the FPU object to become unavailable for further I/O, and delete the volume set continuation block, if it exists.
- Remove_fpu—No operation.
- Unload_fp—Insure that all data structures are deallocated before deleting the function processor image.
- Configure_fp—Not implemented by F11FP. This is an entry for function processors supporting hardware devices.

25.1.4 Mounting a volume

Mount uses the *io$c_ready_fpu* function code to complete the mount process, which includes:

1. Create and initialize volume set continuation data structures (RVT, VCBs), if mounting a volume set
2. Read and check the home block and the volume label
3. Initialize the FPU from the home block
4. Create an object container to hold the file objects for the files to be opened on the volume
5. Request delivery of an AST to invoke mount verification when the LBU(s) layered beneath the F11FP makes a state transition
6. "Open" the index file (this is a simulated OPEN, since the OPEN function can not be invoked until the index file is opened)
7. Open the system files (storage bitmap file, master file directory) and fill out the volume object fields from these files
8. Check that the volume was properly dismounted; if not, perform a rebuild

25.1.5 Dismounting a volume

Dismount uses the *io$c_unready_fpu* function code to complete the dismount process, which includes:

1. Mark the FPU for dismount
2. Wait for all user files to close, then:
    1. Write to the volume SCB indicating the volume was properly dismounted
    2. Close the system files (storage bitmap file, master file directory)
    3. Close the index file
    4. Remove the request for an AST from the LBU layered beneath the F11FP on a state transition
    5. Delete cached file objects
    6. Delete cache tables
    7. Delete the file object container
    8. Delete the volume set continuation data structures, if dismounting a volume set

25.1.6 Volume Sets

MICA supports existing VMS volume sets. Complete volume sets must be mounted; there is no support for mounting incomplete volume sets. The *io$c_add_to_volume_set* function is not supported by the F11FP. Therefore, volume sets cannot be created or expanded in MICA. MICA supports stripe sets that provide the same service as volume sets, as well as more efficient disk space utilization with better load balancing.

25.1.7 Object Names

The FPU object is named by the mount facility. The FPU normally has the same name as the volume name in the volume home block. If an FPU object with the same name already exists, all or part of the name of the LBU beneath the FPU is appended to the name.

File object names are the same as the file's file ID.

25.1.8 Access Matrix

There is an access matrix associated with every open file. The access matrix contains information about the current access modes to the file: read, write, execute, shared read, and shared write. The *io$c_dfile_access* (open) function checks the access and sharing modes of the open request with the file's access matrix. If access to the file is granted, the access mode of the open request is stored in the channel object, and the file's access matrix is updated to include the current access mode. The channel object will not contain a pointer to the file object unless access to the file has been granted. Unshared read access can be overridden by an entry in the volume ACL.

25.1.9 Security

An ACL is associated with a volume object when the object is created. The volume ACL is derived from the VOLACL.SYS file, and is checked whenever a channel is created to the volume.

File protection is managed by access control lists (ACLs) associated with a file object when the object is created. The file object ACL is derived from the file's ACL in the file header. The ACL can be modified (with write privilege) using the *io$c_write_attributes* function.

25.1.10 Mapping & Retrieval Pointers

When a file is accessed, its map pointer block (MPB) is built from the map area of the file header. The MPB contains retrieval pointers that map the virtual blocks of the file to the logical blocks of the volume. Each retrieval pointer describes one file extent. A file extent describes a contiguous group of logical blocks allocated to the file.

25.1.11 Read/Write

Two kinds of read/write functions are processed by the F11FP: paged I/O for memory management and image activation, and virtual I/O. For virtual I/O, the F11FP must build a host transfer list (HTL), which maps the I/O buffer virtual addresses into physical addresses. The F11FP also locks the physical pages into memory before initiating the I/O transfer. The mapping and locking functions are performed by a system service. For paging I/O, the HTL is already built and the pages are locked by the memory management facility.

All file system on-disk data structures are written using a careful write strategy that insures their integrity in the event of a system crash.

25.1.12 Caches

The F11FP uses a caching mechanism to provide fast access to the contents of the following system files:

- Index file bitmap and file headers
- Master file directory and other directories
- Storage bitmap

The F11FP also maintains a cache of deaccessed file objects to avoid the cost of rereading the file headers. Directory file objects, and possibly other file objects, are cached. The cached file objects are available for deletion if system space is needed. No other caches are implemented by the F11FP.

25.1.13 Other Topics

- Cancel I/O

  The F11FP provides for I/O cancellation under the following conditions:

  — When a request is queued to a system thread

  — While waiting for mount-verification to complete

- Condition values

  The F11FP maps the IOSB condition value returned by the lower-layer LBU into its own set of values. Details on I/O status condition values are TBD.

- Mount verification

  Mount verification is implemented on a logical volume basis. The verification procedure checks that the home block matches that of the FPU object. While undergoing mount verification, all operations on the volume are stalled. This feature can be switched off when a volume is mounted.

- Quotas

  Quotas are implemented by a quota file, as described in Chapter 24, Disk File System Function Processors. There is one entry per user containing the user's quota, amount of space used, and overdraft. Quotas can be enabled and disabled on a per-volume basis. \Quotas will not be implemented at FRS.\

CHAPTER 26

RECORD MANAGEMENT SERVICES

26.1 Overview

Mica RMS is a set of generalized library routines that assist user programs in processing and managing files and their contents. The interfaces provided by Mica RMS routines are used to uniformly access files within the defined client-server environment.

Mica RMS is designed to meet several goals:

- Ease of use—This goal is reflected by the user interface design. Mica RMS services are accessed through procedure calls. Each service procedure has a few (less than a dozen) parameters, many of which are optional and default to often-used values. The parameters appearing in the interface are the commonly-used file attributes, the required buffer pointers, and the outputs from the services. For infrequently used input options, the services provide an input parameter, which is an item list. One advantage of using an item list is that the options can be enhanced without affecting the user interface.
- Fast response time—The data retrieval services are designed to minimize run-time decision making.
- Device independence—Mica RMS, like VMS RMS, offers device-independent file handling.
- Modularity—The RMS implementation supports the easy addition of enhancements. For example, supporting a new device type or file organization is fairly straight-forward. Implementation avoids special case code as much as possible.

An overview describing the Mica RMS framework is presented in the following order:

- A list of RMS functions that are available in the Mica system and a list of VMS RMS functions omitted from Mica RMS
- Mica RMS programming interface sampler
- A short note on overall request flow through the Mica RMS services

26.1.1 RMS Functionality

Much of the RMS file processing capabilities are inherited from the unlerlying infrastructure of the Mica I/O subsystem. However, record-level management is provided only through RMS. Mica RMS services operate in user mode and allow user programs to:

- Parse and wildcard file names.
- Specify multiple file organizations (sequential, indexed or relative). At FRS, only sequential files are supported.
- Specify multiple ways to share files and enforce access control to files (shared delete, get, put, update, nil and user-provided interlocking). At FRS, the available support allows multiple processes to read share a single disk file. Also, a file may be shared between a single writer and mutiple readers.

- Specify multiple device types for record access. At FRS, RMS supports I/Os to disk devices only. Paths are also provided for conducting I/O to terminal devices connected to the client systems.
- Specify multiple record formats (fixed, variable, VFC, stream, streamCR, streamLF and undefined).
- Transparently access files through the local file system or through the distributed file system (DFS).
- Specify multiple ways to lock and unlock records. At FRS, there is no support available for record locking.

The following functions are not available at FRS, but are planned for future releases:

- File organization—Indexed and relative files
- File access—Shared write access to disk files
- Record locking—Ways to lock and unlock records
- Transaction logs—Journal file I/O operations

The rest of this section lists the VAX/VMS RMS functions that are not planned to be included as Mica RMS functions.

- Transaction log (journal) of file I/O operations
- Asynchronous I/O operations
- Direct record access to mailboxes or message devices
- Remote file access and task-to-task communication by way of DECnet
- Implicit file spooling
- DECK and EOD checking
- Multiple record streams
- File disposition option submit command file on execution of RMS$CLOSE
- Set date and time for file creation, revision or backup

The I/O subsystem functions not replicated in Mica RMS are:

- $ENTER
- $EXTEND
- $NXTVOL
- $REMOVE
- $SPACE

Further, the following system services are not available through Mica RMS: SYS$RMSRUNDOWN, SYS$SETDDIR, SYS$SETDFPROT, and $WAIT. The undocumented VAX/VMS RMS function $MODFY is also not available in Mica RMS.

26.1.2 RMS Programming Interface

Mica RMS services are provided by a set of user-mode run-time library procedures. The procedures are designed with the two goals of ease of use and flexibility. The RMS user specifies various file attributes to suit the requirements of a particular application. There are two categories of file attributes: the ones that are used for file-level functions (such as Create and Open) and the ones that are used for record-level functions (such as Get and Put). The attibutes are specified to the Mica RMS services by parameters. The attributes appear either as explict parameters, or as options in item lists.

The Mica disk file system assigns a *file_reference* to each file. The *file_reference* can be used to access a file effeciently. The *file_reference* is not identical to the VMS file ID. The Mica file system does not guarantee that a *file_reference* can always be used to access a file. This implies that while accessing or deleting a file, the file specification must always be provided.

Each Mica RMS service returns a completion status. The status codes are not yet specified.

Samples of Mica RMS services are shown below.

In order to to open a file, the user calls the Open (*rms$open*) procedure. The *rms$open* procedure is defined as:

```
PROCEDURE rms$open(
            IN file_name : string(*);
            IN default_file_string : string(*) OPTIONAL;
            IN quick_file_ref_in : rms$file_ref_identifier OPTIONAL;
            IN access_request : rms$file_access_control OPTIONAL;
            IN open_input_options : POINTER exec$item_list_type = NIL;
            OUT file_handle : rms$file_handle;
            OUT file_information : rms$standard_file_info OPTIONAL;
            OUT resultant_file : rms$file_reference OPTIONAL;
            OUT quick_file_ref_out :  rms$file_ref_identifier OPTIONAL;
            ) RETURNS status;
```

The caller specifies the name of the file that is to be opened, using the required input parameter *file_name*. Using the *file_name*, and the optional parameter *default_file_string*, RMS forms a fully qualified file name, which can then be used to access the file. A fully qualified file name has the following format:

```
volume_name:[directory_specification]file_name.type;version
```

A file can be accessed effeciently if the optional input parameter *quick_file_ref_in* is specified together with the *file_name*. The parameter *quick_file_ref_in* contains the file's *file_reference* (as maintained by the Mica file system), and the *volume_object_id* (as maintained by the Mica Object Architecture).

The *rms$open* service returns a status containing the results of the operation, and a *file_handle*. A *file_handle* is a pointer to a data structure that refers to a file context, maintained internally by Mica RMS. The user is required to input the file handle for subsequent data retrieval and management services. The user does not interpret the contents of the file handle.

After opening a file, the caller can perform I/O operations to the file by calling the appropriate data retrieval or data output service. The I/O services are classified according to the record access mode, to shorten access path lengths. The record access mode can be sequential, random by record position (RRP), or random by key. I/O operations to files are also dependent upon file characteristics such as the file organization, record format and device on which the file resides. These file characteristics are set at the time the file is created, and are known when the file is opened. RMS sets up a data retrieval and data output vector according to the file characteristics, for each data access mode. The caller, however, simply invokes the generic retrieval or output procedure, strictly based upon the record access mode.

For example, to get a random record by the record's position from a sequential disk file with variable length record, the user simply calls *rms$get_rrp* procedure, which is a procedure of the *rms$ptype_get_rrp* type. Internally, however, the call is handled by the procedure *get_rrp$seq_dsk_vfc*, which is also of *rms$ptype_get_rrp* type.

Examples of sequential Get and Put services are shown below:

```
TYPE
    rms$ptype_get_sequential: PROCEDURE(
        IN file_handle : rms$file_handle;
        IN in_record_position : POINTER rms$record_position_info OPTIONAL;
        IN move_mode : boolean = FALSE;
        IN in_options : POINTER exec$item_list_type = NIL;
        OUT user_input_buffer : BYTE_DATA(*) CONFORM OPTIONAL;
        OUT read_data_buffer_pointer : POINTER anytype CONFORM OPTIONAL;
        OUT read_data_length : integer OPTIONAL;
        OUT out_record_position : POINTER rms$record_position_info OPTIONAL;
        ) RETURNS status;

    rms$ptype_put_sequential: PROCEDURE (
        IN file_handle : rms$file_handle;
        IN data_output_buffer : BYTE_DATA(*) CONFORM;
        IN in_record_position : POINTER rms$record_position_info OPTIONAL;
        IN in_options : POINTER exec$item_list_type = NIL;
        OUT out_record_position : POINTER rms$record_position_info OPTIONAL;
        ) RETURNS status;
```

To close a file, the caller invokes the Close (*rms$close*) service.

26.1.3 Sample I/O Request Flow

1. An application calls *rms$open* to open a file MYFILE.TXT.
2. The *rms$open* service processes the file name and determines that:
   - The volume name is MYVOL
   - The directory in which the file is to be opened is BETA
   - The file name is MYFILE.TXT
3. The *rms$open* service calls *exec$translate_object_name* with the volume name string MYVOL as input parameter to obtain the FPU object ID.
4. The *rms$open* service calls *exec$create_channel* with the FPU object ID as input to obtain the channel object ID.
5. The *rms$open* service accesses the channel to obtain the necessary security clearance.
6. The *rms$open* service calls *exec$request_io* with input parameters channel ID, IOSB, function code *io$c_dfile_access_file*, the file name with the complete directory path and the file attribute list to access the file.
7. The *rms$open* service builds a file context, and returns a file handle to the user. A data-retrieval vector has also been set up for all I/O operations. The vector entries are (for example):

   ```
   get_seq$seq_dsk_fixed
   put_seq$seq_dsk_fixed
   get_rrp$seq_dsk_fixed
   get_key$seq_dsk_fixed
   put_key$seq_dsk_fixed
   ```
8. At this point, I/O operations can be done on the file.
9. The user closes the file by invoking the *rms$close* service. The *rms$close* service checks that there are no I/Os outstanding on the file, writes out the dirty buffers, and calls *exec$request_io* with the function code *io$c_dfile_deaccess* to close the file. The *rms$close* service deletes the I/O channel by calling *exec$delete_object_id*. The file context area is deallocated, and the pointer to the file context area is initialized to null.

CHAPTER 27

CACHING

27.1 Overview

The purpose of this chapter is to describe the mechanism for caching disk data blocks in the Mica operating system. The chapter is not complete. The Mica system architects recognize the need for data caching to improve application I/O performance. The design of the caching mechanism, however, is deferred until further information is available on the performance of the Mica I/O system.

The purpose of the chapter overview is to summarize the important issues related to data caching in Mica. Mica has a number of features designed to improve system I/O performance that are summarized below. The decision to defer data caching recognizes these performance features as being sufficient for the initial release of Mica, given the available engineering resources and development schedule.

Future work on the data caching design will replace this summary.

27.1.1 Issues Related to the Design of a System-wide Data Cache

- Mica has a number of features in place to reduce the need for a system-wide data cache. For example, memory management caches segment object descriptors and images on the standby page list. The image cache reduces a significant amount of disk I/O related to image activation for programs that are re-invoked during a short time. In addition, RMS also supports multiblock data buffering for sequential read-ahead and write-behind. Multiblock I/O improves application I/O performance by transferring larger units of data on each I/O request.

  Significant I/O performance improvement is also provided in Mica by disk striping. Applications in the compute-intensive, scientific domain using large amounts of data will experience significant burst rate I/O performance improvement because of this feature. New applications can also use the large address capabilities in Mica to map files into the application's address space.

- The initial release of Mica is designed to support Quartz as well as the compute-intensive server environment. Quartz will have to implement a cache that is integrated with the Quartz system recovery techniques. Quartz would have to bypass whatever data cache is implemented in Mica.

- The Mica system architects are also considering future designs for recoverable index files in RMS. The design of recoverable index files leaves no choice but to cache them directly in RMS. Caching in RMS puts the cached data closer to the user of the data and allows greater control over the cache contents. A disadvantage of caching in RMS is the difficulty of sharing cached data amoung separate users.

- The I/O characteristics of key target applications for Glacier exhibit patterns that will not benefit from a system data cache. In particular, applications that read a small data set, perform extensive computation or simulation of discrete events, and write large sequences of event records will not benefit from a data cache.

27.1.2 Summary

The overview summarizes the relevant issues in the decision to defer the design of data caching in the system. Future work on data caching will use performance data from applications running on Mica systems to determine the requirements for data caching.

CHAPTER 28

FILE MANAGEMENT UTILITIES

28.1 Overview

File management utilities are system management tools for managing disk and tape devices. In general, the utilities in Mica support the same functionality as the VMS utilities of the same name. Additional functionality is added to the Initialize and Mount Utilities for managing virtual devices, described in Chapter 15, Direct Access Mass Storage Function Processors.

The file management utilities described in the chapter include:

- The Initialize Utility (INITIALIZE)
- The Mount Utility (MOUNT)
- The Dismount Utility (DISMOUNT)
- The Verify Utility (VERIFY)
- The Backup Utility (BACKUP)

INITIALIZE, MOUNT, and DISMOUNT are completely new Mica implementations. VERIFY and BACKUP are derived from the currently released VMS versions. They are modified to use the Mica file system interface.

28.1.1 Goals

The goals for the Mica implementation of the file management utilities are the following:

- Provide the tools for configuring virtual and logical devices in the system
- Provide a consistent means to make virtual and logical devices available for processing
- Provide a means to validate and archive the data on storage devices

28.1.2 Requirements for the File Management Utilities

The file management utilities must:

- Be callable from the Mica system management environment
- Support the Files–11 ODS-2 standard disk format with ODS2-3 enhancements in INITIALIZE, BACKUP and VERIFY
- Support management of Mica shadow-set and stripe-set virtual disks
- Use VMS BACKUP save-set formats for compatibility with VMS archive media
- Use file placement information in BACKUP /IMAGE save sets to save and restore volumes
- Provide a minimal set of utilities in the off-line system to support system installation

28.1.3 Utilities in the Off-line System

All of the file management utilities are included in the off-line system for system management operations. BACKUP and INITIALIZE are used during system installation to initialize the system disk as a Files–11 volume and to restore the system disk from the installation save set. VERIFY is included in the off-line system to check that all the system disk directories are properly created if the on-line system has problems booting from the new system disk.

28.1.4 Integration with System Management

For the initial release of Mica, the file management utilities are integrated with system management to share common Mica functionality.

The utilities are invoked from client systems running the system management user interface (SMUI). In addition, system management provides a means of communication between a utility and the user on a client system. The utilities communicate with the user to confirm an operation and retrieve a response or to notify the user of special circumstances.

Two utilities require operator communication that is also provided through system management. BACKUP notifies operators when tapes must be mounted to continue a save or restore operation. Operator-assisted MOUNT also notifies an operator to recover from a non-fatal mount error.

28.1.5 Description of the Utilities

The file management utilities are implemented as user-mode library routines. They are called from either system management or the configuration manager.

The utilities issue requests to function processor units (FPUs) in the system to carry out specific operations. The file system and virtual device function processors support operations the file management utilities use to initialize, configure, and manage volumes and logical devices.

The following sections provide a brief description of the file management utilities.

28.1.5.1 The Initialize Utility

INITIALIZE invokes the appropriate function processor to initialize either a logical block device as a volume, or a group of logical block devices as a set of members of a virtual device.

There are three types of INITIALIZE operations:

- Initializing a Files–11 ODS2 volume
- Initializing a shadow-set virtual device
- Initializing a stripe-set virtual device

The type of INITIALIZE operation is determined by a *function_code* parameter to the utility.

INITIALIZE creates a Files–11, shadow-set, or stripe-set FPU for the new volume or virtual device. The parameters to the utility are packaged into an I/O request to the FPU. The utility issues the respective initialization request for the volume or virtual device. The function processor creates the on-disk data structures that establish the logical block unit as a volume, or as a member of a shadow set or stripe set.

After the volume or virtual device is initialized, INITIALIZE optionally *mounts* the volume or device for further processing.

28.1.5.2 The Mount Utility

MOUNT makes disk and tape volumes available for user access. In addition, MOUNT creates virtual devices in the Mica I/O hierarchy.

There are six types of MOUNT operations:

- Mounting a Files–11 ODS-2 volume
- Mounting a remote Files–11 volume (using DFS)
- Mounting a shadow-set virtual disk
- Mounting a stripe-set virtual disk
- Mounting a magnetic tape as a foreign device
- Mounting a disk as a foreign device

The type of MOUNT operation is determined by a *function_code* parameter to the utility.

All the FPU objects created by MOUNT are entered into a system-level object container. This means that mounted devices are visible system-wide. Access to mounted devices may be restricted by using access control lists (ACLs).

MOUNT determines what type of entity the mount operation is trying to mount, for example, a volume set or a virtual device. The utility parameters are checked for consistency and the mount device or units of a virtual device are checked to see if they exist.

If a volume is being mounted, MOUNT creates a Files–11 FPU for the volume and brings the volume FPU ONLINE for further processing.

If a virtual device is being mounted, MOUNT creates a shadow-set or stripe-set FPU for the virtual device and brings the virtual device ONLINE for further processing.

Foreign devices are mounted by allocating the logical block unit and ensuring that the FPU is ON-LINE for user access.

28.1.5.3 The Dismount Utility

DISMOUNT makes a volume or virtual device inaccessible. The utility issues a request to a volume or virtual device FPU to not accept any new I/O requests and deletes the FPU after all current processing completes.

28.1.5.4 The Verify Utility

VERIFY checks the on-disk structure of the Files–11 ODS-2 volume and validates the directory organization. In addition, VERIFY can rebuild the storage bitmap and recover lost files. The volume should be mounted with exclusive access to ensure there is no concurrent file activity. VERIFY does not check that the volume is mounted with exclusive access.

VERIFY follows a sequence of eight phases to validate the index file, storage bitmap, directory structures, and quota file on a volume. The utility runs entirely in user mode. All reads and writes to the volume go through the file system function processor for virtual block I/O.

VERIFY currently does not check for consistency across extension headers for sparse file allocation. VERIFY requires enhancements to support ODS2-3 modifications for file reference counts.

28.1.5.5 The Backup Utility

BACKUP runs entirely in user mode within the system management process. All access to files and directories on volumes is through I/O requests to the file system function processor. Access to magnetic tape archive media is provided by a TMSCP function processor, described in Chapter 16, Magnetic Tape Function Processors. The file system and tape function processors are part of both the on-line and off-line system. Therefore, BACKUP does not require special functionality to run in the off-line environment.

BACKUP uses the system management services to send confirmation messages to the user and to receive responses. The utility also communicates through system management with operators for assistance in mounting devices.

VMS BACKUP currently saves file placement information during an /IMAGE save operation, but does not use the information when restoring the volume. The Mica implementation will use placement information during /IMAGE restores to support off-line archiving of database relations.

Image Related

This set of chapters describes the image-related components of Mica.

CHAPTER 29

OBJECT MODULE AND IMAGE FILE FORMAT

29.1 Overview

29.1.1 Requirements

There are three requirements for this chapter:

- The chapter specifies the image file format required by the image activator.
- The chapter specifies the object module format required by the linker and the object module loader.
- The chapter specifies the generic module format required by the librarian.

The first two requirements are related to each other because object modules share the same format as image files. The first two requirements are related to the third requirement because object modules and image files are examples of the types of files maintained in libraries.

These formats were designed for the following goals:

- The format allows clean extensions in future releases to add functions.
- The format allows for files greater than four gigabytes.
- The module format allows different types of modules to be mixed in one library.
- The object module format allows object modules to be run without linking.
- The object module format allows separate object modules to be combined into one object module.
- The image file format allows fast image activation.

The increasing division of programs into separate object modules leads to an increasing number of object modules, with a concomitant increase in overhead during linking. Combining separate modules into a single module is a means of controlling this overhead.

The tendency in VAX/VMS has been to add functionality to object modules and image files, resulting in larger files. If this trend towards larger files continues, it will become desirable to separate the different parts of an object module or image file into separate files, and yet maintain them in a common library. A common module format allows for this separation. A common module format also allows for a single implementation of the librarian.

The data structures in this chapter allow for both 64-bit addresses, and file sizes greater than four gigabytes. Systems that do not support 64-bit addresses, or file sizes greater than four gigabytes, must have the upper longword of these fields zero to ensure that the values stored in the fields are valid 64-bit values.

29.1.2 Description

An object module and an image file are both examples of a module. A module can define names, refer to names, and be in a library. An object module and an image file share additional similarities because both can be activated, and because image files are created from object modules.

The primary users of object modules are compilers, the linker, and the loader. The primary users of image files are the linker, the debugger, and the image activator. Generally, references to the linker refer to both the linker and the loader. The loader is activated when an object module is run without first linking it.

All modules have a common header format and a common name table format so that a common librarian utility can be used for different types of libraries. All modules contain within their header an index to their different sections. Different types of modules may contain different types of sections, but the module header contains an index that provides a means of accessing sections specific to a module type.

Module specific sections that are common to both object modules and image files are:

- Global symbol table
- Debug symbol table
- Entity consistency check table
- Data sections
- Code sections

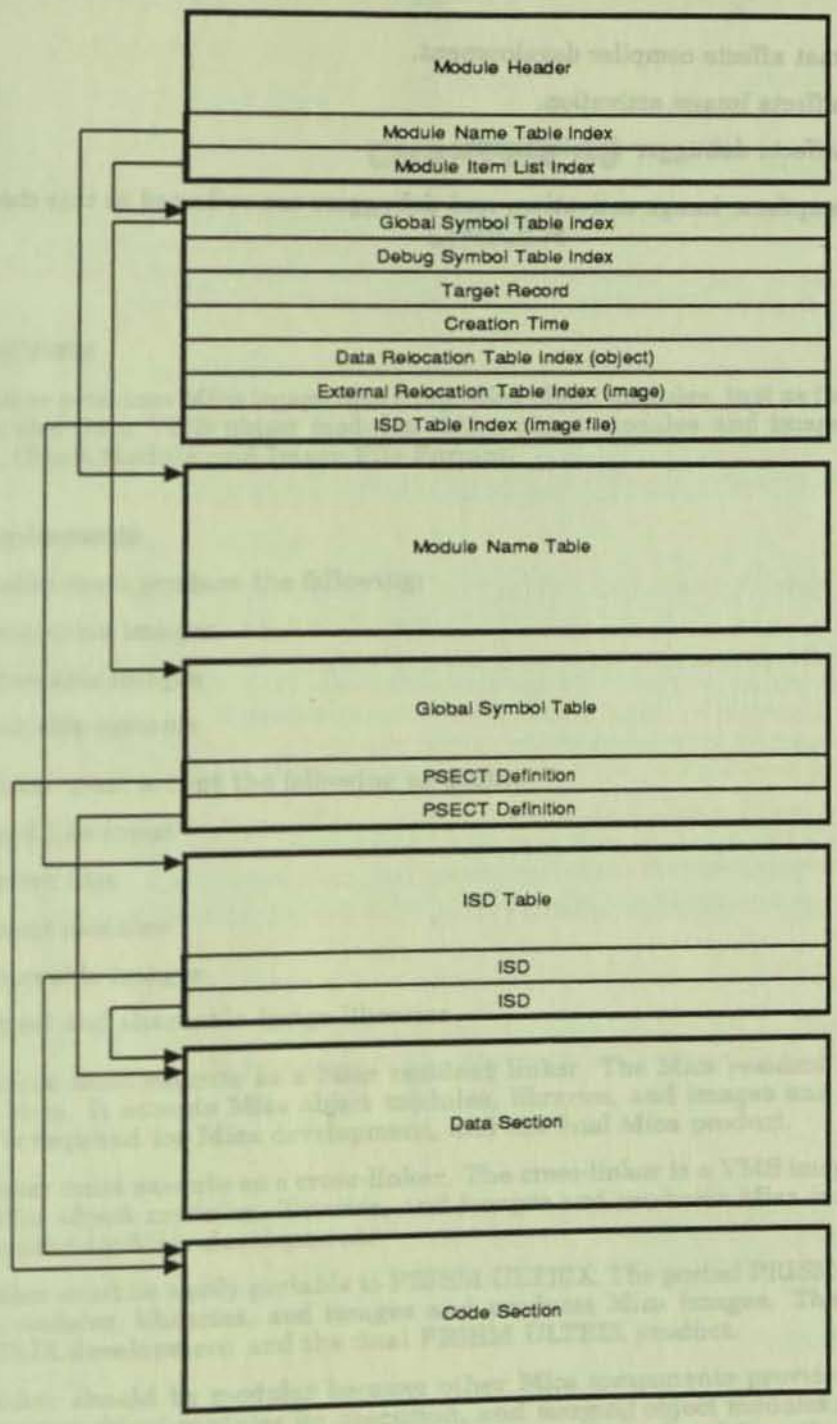
The object module specific sections are:

- Linker directive table
- Data relocation table

Image file specific sections are:

- Image section descriptor table
- Image relocation tables
- Activation tables
- Transfer vector table
- Debug module table

Figure 29–1: Object Module and Image File Format



29.1.3 Dependencies

- Object module format affects compiler development.
- Image file format affects image activation.
- Image file format affects debugger development.

The requirements of compilers, image activation, and debuggers are reflected in this chapter.

CHAPTER 30

LINKER

30.1 Overview

The Mica linker produces Mica image files from Mica object modules, just as the VMS linker produces VMS image files from VMS object modules. Mica object modules and image files are described in Chapter 29, Object Module and Image File Format.

30.1.1 Requirements

The Mica linker must produce the following:

- Mica executable images
- Mica shareable images
- Mica bootable systems

The Mica linker must accept the following as input:

- Command line input
- Mica option files
- Mica object modules
- Mica shareable images
- Mica object and shareable image libraries

The Mica linker must execute as a Mica resident linker. The Mica resident linker is a Mica image running on Mica. It accepts Mica object modules, libraries, and images and produces Mica images. This linker is required for Mica development, and the final Mica product.

The Mica linker must execute as a cross-linker. The cross-linker is a VMS image running under VMS. It accepts Mica object modules, libraries, and images and produces Mica images. The cross-linker linker is required for Mica development.

The Mica linker must be easily portable to PRISM ULTRIX. The ported PRISM ULTRIX linker accepts Mica object modules, libraries, and images and produces Mica images. This linker is required for PRISM ULTRIX development and the final PRISM ULTRIX product.

The Mica linker should be modular because other Mica components provide overlapping functions, such as preparing object modules for execution, and merging object modules into one. Modular code also allows the code dependent on the operating system to be isolated from the bulk of the Mica linker code, which in turn allows the cross-linker and the Mica resident linker to be essentially the same.

The linker should resolve symbols and cluster program sections compatibly with VMS.

30.1.2 Implementation

During its execution, the Mica linker maintains a number of internal data structures, examples of which are:

- Table for global symbols
- Table for PSECTs
- Table for environments
- List for unresolved symbols

The Mica linker is implemented in two passes which, together with the work before and after each pass, define five linker stages: initial, pass 1, intermediate, pass 2, and final.

30.1.2.1 Initial Stage

The goal of the initial stage is to understand the command line and the commands contained in the option files. The linker associates its input files with the clusters that have been specified. (Clustering is a means of explicitly grouping modules or PSECTs so that the resulting memory will tend to be together.)

30.1.2.2 Pass 1

The goal of pass 1 is to read all of the symbol and PSECT information in the input files and to maintain the specified clustering. During pass 1, the linker reads all its input files. The commands in the linker command tables in object modules are performed. New clusters are created if needed. Symbols and PSECTs described in object modules and shareable images are organized in the internal tables. Symbols referenced by object modules are either resolved to an already defined symbol or are added to the unresolved symbols list. An unresolved symbol can be resolved by a module that is:

- Explicitly included in a linker command
- In the currently accessed library
- In a library implicitly or explicitly specified after the currently accessed module

30.1.2.3 Intermediate

The goal of the intermediate stage is to prepare for reading the data and code from the input files. The linker calculates the virtual memory requirements of each segment of the image from information read during pass 1.

30.1.2.4 Pass 2

The goal of pass 2 is to create the image file with the data and code sections of the object modules, and to fix up all the relocations within it. During pass 2, the linker copies the data and code sections from the object modules to their location in the image file. The linker also manages the following during pass 2:

- Map file information
- Fix-ups
- Demand zero compression
- The image's header
- The image's image section descriptor table
- The image's activation tables

- The image's transfer addresses
- The image's relocation tables
- The image's global symbol table
- The image's debug symbol table
- The image's target system \Each object module contains targeting information that the linker must check\
- The image's use of vector instructions \Each object module contains information on the use of vector instructions that the linker must check\

30.1.2.5 Final Stage

The goal of the final stage is to write the image file to disk and report the statistics of the link.

30.1.3 Compiler Dependency

The Mica linker shares many functions with compilers; both must:

- Maintain tables for identifiers
- Read and write modules (Definition modules, object modules, and image files all share common format.)
- Handle errors and text in a method that allows for internationalization
- Maintain performance statistics
- Manage large amounts of memory

To speed development and ease future support, the Mica linker utilizes the DECwest compilers' compiler shell and super shell; these are the parts of the DECwest compilers that implement the above functions. The super shell provides operating system functions, and the compiler shell provides compiler functions. Both are described in *Compiler Shell Documentation for the Pillar and C Compilers*.

The Mica linker utilizes the following modules from the DECwest compilers' super shell:

- I/O
- Memory management
- Error reporting
- System information

The Mica linker utilizes the following modules from the DECwest compilers' compiler shell:

- Text handling
- Generic table
- Identifier table
- Performance statistics

The linker accesses these modules by linking with the compiler shell and super shell shareable images.

CHAPTER 31

IMAGE ACTIVATION

31.1 Overview

The linker produces an executable image as the end product of program development. During process creation, the thread creating the process specifies the image to be executed by the new process. After the creation of the process and the initial process thread, the image file is mapped into the newly created address space. This mapping occurs in the context of the initial process thread.

Image mapping involves several steps that prepare the image for execution. The image activator opens the image file, thereby establishing a channel to obtain the necessary information to map the file. If the image does not already have an associated segment object, the image activator creates a segment object for the image, building prototype PTEs for the image file. The image activator maps the image into the user's address space, resolves certain address references, and establishes the debugger and traceback handlers.

31.1.1 Goals/Requirements

The Mica image activator has the following goals:

- All images are automatically and transparently shared among all users.
- Optimal performance is achieved by issuing a minimal number of disk reads to initially map the image and delaying most fixups by delaying the loading of shareable images.

31.1.2 Functional Description

31.1.2.1 Image Initialization

No special code exists in Mica to read images into memory for initial execution. Instead, the paging mechanism is used to "page" an image into memory. The image activator configures the process page tables to reflect all pages in the image file.

Mica performs the following steps to support image activation:

1. Opens the image file

   The image activator issues a read-only share open service on the image file. This service returns a channel ID to the file.

2. Creates a section

   The image activator calls the *exec$create_section* system service. The caller specifies the channel ID from the previous system service call, and a mapping type of *e$k_image_map*. This service returns a *section_id*.

3. Maps the section

   The image activator calls the *exec$map_section* system service, specifying the *section_id* returned from the *exec$create_section* system service. This service returns the starting and ending addresses that delimit the mapped image in the virtual address space.

4. Performs fixups

   The starting address identifies where the image's image header begins. The image activator examines the image header, and performs the necessary image fixup operations on the image.

5. Handles message sections

   If any message sections are present in the image, as indicated by the image header, the image activator calls the routine to add these message sections to the process. The nature and functions of this routine are described in Chapter 3, Status Values, Messages, and Text Formatting.

6. Maps shareable images marked "activate immediately"

   The image activator examines the image header, and maps any shareable images which are marked "activate immediately". The image activator performs the external fixups for those shareable images once they are mapped. Note that this is a recursive call to the image activator.

7. Calls initialization procedures

   Once the image activator maps and fixes up the "activate immediately" images, it examines the image header, and calls any initialization procedures at their specified entry points. These initialization procedures provide the functionality of the LIB$INITIALIZE routine in VMS. The image activator does not guarantee the order between images of initialization procedure calls, but it does guarantee each procedure is called only once before the user executes any code within that shareable image.

8. Invokes image

   After the image activator has invoked all initialization procedures, it calls the image at its transfer address.

31.1.2.2 Image Exit

When an image returns to the image activator, the *exec$thread_exit* system service is issued, which begins thread termination. The *exec$thread_exit* system service simply calls each exit handler that has been declared by the thread, and then invokes the *exec$delete_thread* system service. If the process has other threads executing, those threads continue to execute normally. Image exit occurs when the last thread in the process exits and the mapping objects and section objects for the image are deleted during object container rundown.

31.1.2.3 Autoload Procedure

The autoload procedure operates similarly to the "activate immediately" method described above. The autoload procedure loads shareable images and resolves the external references when the reference is encountered, rather than at initial image activation time. This reduces the overhead of initial image activation, and maps shareable images only when they are actually required.

31.1.2.4 Installation of Images

The Install Utility serves two purposes. It provides for the installation of a shareable image:

- Within the shareable image space
- With the WRITE attribute

The Install Utility creates a section object for the image, which causes a segment object to be built. The segment object has a "system channel" to the image which implies that the image is effectively installed "opened".

31.1.2.5 Images Within Shareable Image Space

When a shareable image is installed in the shareable image space by use of the /BASE qualifier, the Install Utility opens the image file, and creates a segment causing prototype PTEs to be built. The segment object for the shareable image contains the base address for the image within the shareable address space. When the shareable image is loaded, it is mapped at the specified address. If the image cannot be mapped at the specified address due to addressing conflicts, an error is returned, and the shareable image is not mapped.

When the shareable image is installed no fixups, internal or external, are performed. Note, however, that since no external fixups are performed, any referenced shareable images are treated just like referenced external images. This allows later versions of shareable images to be installed at different base addresses while the system is running, and the latest image is properly loaded.

Images installed in shareable image space may reference other images though use of the autoload capability or the activate immediately capability. These referenced images do not need to reside in shareable image space.

31.1.3 Issues to be Resolved

- Exact detail of message section addition. This is dependent on the design of message sections and the definition of the routines.

System Management and Administration

This set of chapters describes the components of Mica relating to system management and system administration.

CHAPTER 32

SYSTEM MANAGEMENT

32.1 Overview

This chapter provides an overview of Mica system management. First, the chapter lists the system management functions that a system manager or operator can perform. Second, the chapter describes a model for system management. Third, the chapter describes how system management on multiple Glacier or Cheyenne systems is implemented. Next, the chapter briefly describes system security, and discusses various files that the system management server maintains. Finally, the paper presents some considerations for the design of the server.

32.1.1 Functional Description

Mica system management allows an operator or system manager to do the following:

- Maintain user accounts.
- Provide user account information for authentication.
- Maintain proxy accounts.
- Provide proxy account information for authentication.
- Define and maintain identifiers.
- Set up and modify access control lists (ACLs).
- Access file utilities (BACKUP, MOUNT, DISMOUNT, INITIALIZE, VERIFY).
- Access the configuration manager.
- Provide Distributed File System (DFS) management.
- Maintain system parameters.
- Set up system level logical names.
- Access diagnostics.
- Access the Monitor utility.
- Set up and modify the system startup facility.
- Access the Error Report utility.
- Issue logging commands.
- Maintain RPC binders.
- Access the User Environment Test Package (UETP).
- Display system information.
- Install software.

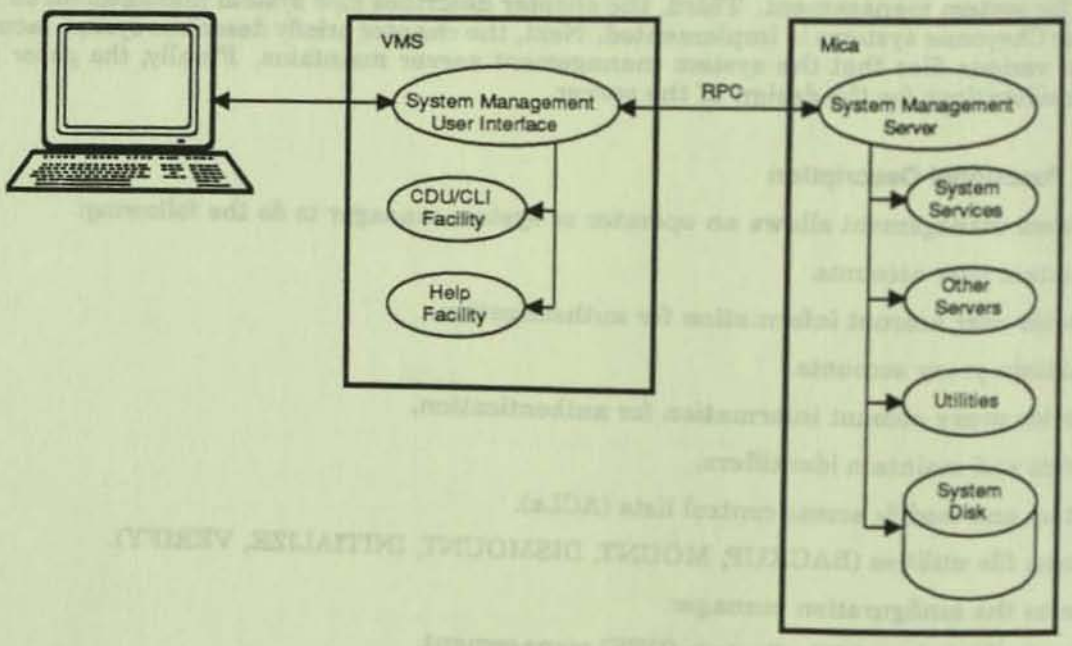
- Manipulate log files.
- Access the Install (Images) utility.
- Access Quartz utilities.

32.1.2 System Management Model

The design of system management separates the user interface from the functional implementation. The system management interface (SMUI) resides on a client system (a VMS client at FRS). The system management server component resides on a Mica system, and the two components communicate by means of RPC.

By implementing the user interface on the client system, the user interface can take advantage of the client software on which it is implemented, and can also be explicitly tailored to a particular client. Figure 32–1 shows these two components.

Figure 32–1: System Management Components



The following sections describe these two components.

32.1.2.1 The System Management User Interface

The system management user interface (SMUI) is the client component of the system management software. SMUI provides two interfaces to system management for the system manager or operator: an interactive DCL-like command line interface and an interactive DECwindows interface. SMUI also provides a batch-oriented DCL-like command file. If the system manager has a bit-mapped terminal, both the command line and DECwindows interfaces are available. The DECwindows interface is an easy-to-use, user-oriented interface. If the system manager has a character-cell terminal (or hardcopy terminal), only the command line interface is available. For FRS, the interfaces will be available from a VMS client only, and will exist as a layered product on the client.

The DECwindows interface is layered on top of the command line interface. The DECwindows interface therefore creates a command line.

SMUI depends on the following VMS facilities:

- Command Definition utility (CDU)

- Command Language Interpreter (CLI) routines
- Help librarian routines (LBR routines)
- DECwindows, DECtoolkit, and the User Interface Language

32.1.2.2 The System Management Server

The system management server is the server component of the system management software. The system management server is the only component on Mica with which SMUI communicates.. The server performs all necessary functions, calls, and utility invocations, and sends any responses back to the client interface. The server may also consist of more than one server. (For more information, see the remainder of this section and Section 32.1.8).

SMUI and the server software communicate by means of RPC. This interface resembles a system service interface.

The system management server resides on both Glacier and Cheyenne systems, although the functionality provided for each system may differ.

The system management server has three main functions:

- Accepting calls from other Mica components

  The interface into the system management server are in system service format; the underlying RPC mechanism is not visible to the caller. This interface will not be available for customer use by FRS. However, the interface will be available for internal use. Examples of calls used by other components of Mica include calls for accessing authorization and security information.

- Invoking Utilities

  The system management server invokes callable utilities, for example, the Backup utility. The server passes a parameter list to the utility.

- Calling Other Servers

  The system management server calls other servers, at the request of SMUI. For example, to perform local DFS management, SMUI requests the system management server to call the DFS server. In this case, SMUI calls the system management server, which, in turn, makes a local RPC connection to the DFS server.

32.1.3 RPC Interface

Requests to the system management server are accomplished by RPC. The system management server will use RPC for both intra-node (from Mica facilities) and inter-node (from SMUI) communications.

System management requires that the system management server be able to call procedures in the client while the original server call is outstanding. This functionality is required to implement features such as /CONFIRM mode in utilities.

System management will also use the following RPC functionality: disconnect notification, and a server acting as a client of another server.

32.1.4 Managing Multiple Systems

The following two sections describe considerations for managing multiple Glacier or Cheyenne systems.

32.1.4.1 Glacier Systems

A system manager can manage multiple Glacier systems from one client. However, the system manager can only issue system management commands to one Glacier system at a time.

32.1.4.2 Cheyenne Systems

Multiple Cheyenne systems in the Quartz environment will be managed as separate systems. Thus, to perform a function on more than one system, the system manager must issue a command to each system.

\This plan could change if Quartz requires that one command be issued for all systems. In either case, if Quartz requires any coordination in managing multiple Cheyenne systems, system management will be performed through SMUI. There will be no coordination between system management servers on separate systems, nor will the servers need to communicate with each other.\

32.1.5 Security

Access to system management services is controlled by the proxy access to a Mica server. For each client from which full system management is performed, a specific account is designated on the client. The node and account are entered in the proxy file of the Mica server, and the corresponding account on the server is then granted the identifiers needed to perform system management.

Currently, there are no plans to totally restrict access to the system management server. A process that has been granted access to a Mica server has the ability to access any server on the system. Lack of held identifiers will prohibit the unauthorized user from performing most system management functions.

32.1.6 Subset System Management Access

The set of functions visible to a user on a client may differ, depending on whether the user is performing system management from a system-management or a non-system-management account.

A person on the client (for example, an operator or a user) can access a subset of system management by running SMUI from an account on the client that is different from the system management account. The client node/account proxy pair corresponds to an account on the server that has a subset of system management identifiers (or in the case of a user, may have no special identifiers). The user's identifiers allow access to some functions of the system management server. Functions that a user can perform include setting ACLs on personal files, showing objects, and issuing certain SHOW commands such as SHOW SYSTEM.

32.1.7 Authorization, Proxy, Identifier, and Startup Parameter Files

The system management server maintains the user authorization file, the network proxy file, the identifier file, and the startup parameter file. The system management server is the owner of these files, and is the sole accessor of these files.

The system management server also maintains SYSGEN parameters that are stored in the system image. These parameters are available to the system directly.

Since indexed files are not available at FRS, system management will provide its own keyed access to the information in these files. This may involve creating indexes at system startup, caching the files and/or indexes in memory, and combining files together.

32.1.8 Server Design Considerations

The system management server will be multithreaded, so that multiple server requests can be serviced at the same time.

32.1.9 Issues

The following system management issues are still outstanding:

- In what language should the client software be implemented? The obvious choices are Pillar and C. How much Pillar support will exist on VMS as a target operating system (messages, etc.)? Pillar would be a good implementation language simply for speed of coding and debugging. C is a good choice because of portability to Ultrix; however, it will lengthen implementation time, and it is questionable how much of our client code will be portable to Ultrix, since much of the code will be VMS-specific.

CHAPTER 33

OPERATOR COMMUNICATIONS

33.1 Overview

This document provides an overview of the operator communications facility for Mica (OPCOM). OPCOM provides 2-way operator communications, as well as terminal and file logging capabilities for various types of messages on Mica. These messages include operator messages, error messages, security messages, and account messages.

The first two sections provide a functional description of OPCOM and describe the various OPCOM components. The next section addresses support for OPCOM within the Applications Integration Architecture (AIA), followed by a section on native mode OPCOM calls. Finally, any outstanding OPCOM issues are addressed.

33.1.1 Functional Description

OPCOM provides the following functions:

2-way operator communications. With 2-way operator communications, the following is possible:

- A user can enter an operator request from either a Mica program or a client terminal.
- From a client terminal running system management software, an operator can reply to a request.
- From a client terminal running system management software, an operator can display a list of outstanding operator requests.

Operator message capability. A client user or operator, or a Mica system facility, can send a message to the Mica operator.

General message capability. The general message capability allows an operator to do the following:

- Enable a client terminal to receive operator and/or security messages
- Enable a local Mica file to receive operator, security, error, or accounting messages
- List all log files and operator terminals in use by a server
- Enable or disable the error and accounting function processor units (FPUs). (For information on FPUs, see Section 33.1.2.5).

Broadcast capability. If there is an operator terminal enabled on the client system, the broadcast function allows an operator or system facility to broadcast to terminals on a client system.

33.1.2 OPCOM Components

OPCOM consists of several components. Some of these components reside on the client, while others reside on the server.

The OPCOM components on the client are:

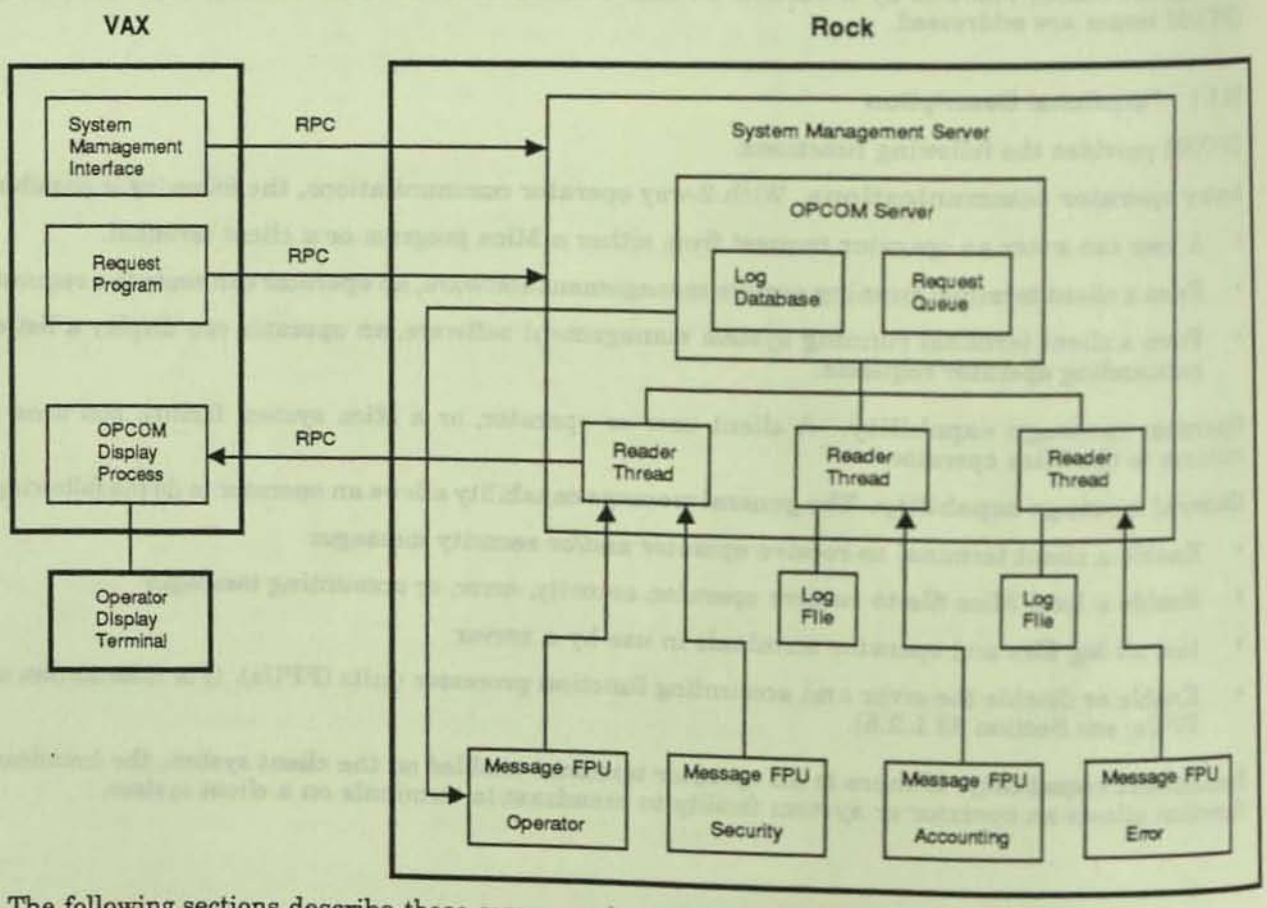
- The client system management interface
- A client operator terminal display process
- The client operator request program

The OPCOM components on the server are:

- The OPCOM server
- Mica message FPUs
- Reader threads of message FPUs

Figure 33–1 shows the relationship of these components.

Figure 33–1: Relationship of OPCOM Components



The following sections describe these components.

33.1.2.1 Client System Management Interface

The client system management interface allows the operator or system manager to enter OPCOM commands. Table 1 describes these commands.

Table 33–1: Client System Management Interface Commands

Command	Description
START/LOGGING	Starts logging of operator, security, error, and account messages. Logging may be specified to a terminal on a client system for operator and/or security messages, or to a file on the server for operator, security, error, and account messages.
STOP/LOGGING	Stops logging of operator, security, error, and account messages
SHOW LOGGING	Shows system logging
REPLY	Replies to an operator request
REQUEST	Makes a request to an operator
SHOW REQUESTS	Shows the queue of unanswered operator requests
ENABLE/DISABLE FPU	Enables or disables error and account FPUs
SEND	Broadcasts a message to client terminals

The client system management interface communicates to the system management server by a remote procedure call (RPC) interface.

33.1.2.2 Client Operator Display Process

An operator display process is started when an operator enters the START/LOGGING request through the system management interface. The display process is connected, by RPC, to an OPCOM reader thread. (For information on reader threads, see Section 33.1.2.6).

The display process does the following:

- Continually reads messages from the OPCOM reader thread until terminated
- Displays the messages on the operator terminal
- Forwards Mica broadcast messages (such as server shutdown messages) to the client broadcast facility

33.1.2.3 Client Operator Request Program

OPCOM provides a request program as part of both the VMS and ULTRIX client software. The program allows a user to make an operator request. This program is a standard MICA program (run by some means such as PRUN REQUEST). The request program, when running on MICA, calls an AIA operator communication routine to make an operator request. This operator request program is in provided to allow a user (other than an operator or system manager) to make an operator request. See Section 33.1.3 for more information on the AIA operator functions.

\This capability is not provided for Cheyenne clients. There is no AIA interface on Cheyenne, and there should be no need for a user to make operator requests. A Cheyenne operator can still make operator requests from the system management interface.\

33.1.2.4 OPCOM Server

The OPCOM server runs as a protected subsystem on the server.

\The OPCOM server may be the same free running process as the system management server\

The functions of the OPCOM server include maintaining a database of logging terminals and files, maintaining a database of outstanding operator requests, and starting and stopping reader threads. (For information on reader threads, see Section 33.1.2.6).

The OPCOM server responds to the commands that are entered through the client system management interface. The OPCOM server responds to these commands in the following way:

- When the START/LOGGING or STOP/LOGGING command is entered, the OPCOM server starts or stops a reader thread and updates a database of log files and terminals.
- When the SHOW LOGGING command is entered, the OPCOM server reports the database of log files and terminals.
- When the REPLY or REQUEST command is entered, the OPCOM server writes the request or reply to the operator message FPU and updates a queue of outstanding operator requests. Also, the OPCOM server periodically scans and rewrites outstanding requests to the operator message FPU.
- When the SHOW REQUESTS command is entered, the OPCOM server reports the queue of outstanding operator requests.
- When the ENABLE/DISABLE FPU command is entered, the OPCOM server sets the specified FPU state to ONLINE or AVAILABLE.
- When a SEND command is entered, the OPCOM server writes the message to the operator message FPU. The message is targeted for the first display process on each client, which, in turn, forwards the message to the client broadcast facility.

All requests to OPCOM are logged to the operator message FPU. This means that the operator message log includes a history of commands processed by OPCOM, including operator requests and replys.

33.1.2.5 Mica Message Function Processor Units (FPUs)

Message FPU's are the mechanism provided in MICA to pass messages from one process to another. There are four message FPU's in MICA that are used to pass system messages. These four FPU's are the operator, security, error, and account message FPU's. Messages are written to the FPU's by various system components, including OPCOM itself. The messages are read by threads of OPCOM, and may be read by other system facilities (such as an error detection program reading the error messages).

Reader threads register with a message FPU before reading messages from the FPU. A thread can register to read a subset of messages in an FPU. For example, a thread can register with the operator FPU to read only tape operator messages. There will be support in the START/LOGGING command to specify a subset of messages for a reader to read.

33.1.2.6 Reader Threads

A reader is a process that reads messages from one or more message FPUs and writes the messages to a file or terminal. The readers of the system message FPU's are threads of OPCOM. A reader thread exists for each START/LOGGING command that is issued. The reader thread registers with the appropriate message FPU(s), and either opens a file or establishes an RPC connection with a display process on a client. Each message read is either written to a log file or sent to the client display process. A reader thread is terminated when OPCOM receives a STOP/LOGGING command or when a client display process terminates.

The messages, as they are read from an FPU, are in binary format. After a reader thread reads a message, further processing is required to translate the message into a meaningful format.

In the case of operator and security messages, the binary message is translatable directly into an ASCII message. The reader thread is responsible for this translation, which is accomplished by looking up the message number in a file. Thus when the reader thread writes the operator or security message to a file and/or a client display process, it is writing a fully formated ASCII message. These ASCII operator and security messages may be mixed when written to a terminal or to a file, and one reader can read and translate both security and operator messages.

\The exact method used to translate operator and security messages to their ASCII counterpart is not defined\

In the case of error and account messages, the reader thread does not translate the message. The message is in binary format, and can be written only to a file (not to a terminal). The error and account messages cannot be mixed with any other messages in a file. The error report analyzer must be run to create error reports from an error file. There will be no support at FRS to generate reports from information in an account file.

33.1.3 AIA Functionality

There will be support in AIA for operator functions (specifically REQUEST and REPLY functions). The operator functions will be synchronous only. This means that a program that makes an operator request and expects a reply will be in a wait state until the reply has occurred. These operator functions are used in the REQUEST program provided with VMS and ULTRIX clients.

A user can access the AIA operator functions through a MICA program, and can thus perform an operator request from a user written program.

This AIA functionality is not available on Cheyenne clients.

33.1.4 Native mode OPCOM calls

The native mode service calls within OPCOM are available for use internally by system utilities such as BACKUP and MOUNT, as well as by the system shutdown facility. These calls will not be documented for external use at FRS.

33.1.5 Manipulating log files on Cheyenne

Since there is no DFS support on Cheyenne, system management must be able to perform some file functions on log files residing locally on the system. System management will provide specific functions to manipulate Cheyenne log files, such as COPY, DIRECTORY, and DELETE. The log files will reside in a known area on the Cheyenne system.

33.1.6 Issues

The following OPCOM issues are still outstanding:

- Will Mica logging be able to support Decnet logging?
- Will Mica be required to provide operator interface and terminal display facilities on ULTRIX clients?
- Does the user REQUEST program need to be provided on a Cheyenne client system?
- Does AIA functionality need to include broadcast requests?

  Probably not.
- How is security handled between the client and the server?

CHAPTER 34

CONFIGURATION MANAGEMENT SOFTWARE

34.1 Overview

This overview summarizes the design and function of the Mica configuration management software. This software consists of three components:

- An error-monitor process, running in user mode
- A configuration-manager process, running in user mode
- A configuration function processor, running in kernel mode

(For the rest of this overview, the term *monitor process* refers to the error-monitor process, except where otherwise specified. Likewise, *manager process* refers to the configuration-manager process, and *function processor* refers to the configuration function processor.)

34.1.1 Goals

The configuration management software has the following goals:

- To provide sufficient performance for Cheyenne requirements
- To autoconfigure physical devices at system boot time
- To autoconfigure new devices into a running system upon request
- To mount disks at system startup
- To reconfigure processors and devices upon request
- To provide required availability for Mica by monitoring error events; detecting imminent device failures; and when possible, reconfiguring hardware

34.1.2 Functional Description

The configuration management software performs specific functions at boot time and during normal operation.

34.1.2.1 Actions at System Boot Time

At system boot time, the configuration function processor performs the following tasks:

- Reads the processor configuration from the RPB (restart parameter block), then stores the configuration.
- Reads module information from the RPB—if the processor provides such information—then stores it.
- Notes vector processor modules.
- Reads the list of bad memory pages that were found during booting, then stores the list.
- Checks on the XMI bus to detect all adapters and controllers.

After the configuration function processor has found all adapters and controllers, the configuration-manager process starts. This manager process then performs the following tasks:

- Reads all information found by the configuration function processor, then writes this information to a database in memory. It also writes this information to the error-log message FPU, so that the configuration of the system is available through the Error Log Report Generator (Chapter 23, Error Logging).
- Reads an exclusion file from the system disk; this file tells which devices are not to be configured into the system. The exclusion file is a fairly static file to be managed by the system manager. It would contain, for example, disks that the system manager did not want automatically mounted at system startup. It is never updated automatically as a result of operating system actions, such as configuring a shadow set.
- Creates a notification message FPU, creates a channel to it, then registers on that channel.
- Loads function processors for devices that are to be part of the configuration, if these function processors were not previously loaded during booting.
- Calls the appropriate function processors to create FPUs that represent the physical devices.
- Calls the function processors for the Mass Storage Control Protocol (MSCP) and the Tape Mass-Storage Control Protocol (TMSCP) to find all disks and tapes on configured controllers.
- Reads from the notification message FPU all disks found by MSCP (See Chapter 15, Direct Access Mass Storage Function Processors).
- Mounts disks not excluded by the exclusion file.

Like the configuration-manager process, the error-monitor process is started after all disks and controllers are found. The error-monitor process creates a channel to and registers with the error-log message FPU, then issues a *read* for device error messages.

34.1.2.2 Actions During Normal Operation

The actions described in this section occur during normal operation—that is, after the system is booted.

34.1.2.2.1 The Error-Monitor Process

During normal operation of the system, the error-monitor process reads error events from the error-log message FPU. The process detects and reports device failure events to the configuration-manager process. For example, it reports when a device has a number of recoverable errors that exceeds a threshold or when a device has incurred a fatal error.

34.1.2.2.2 The Configuration-Manager Process

During normal operation of the system, the configuration-manager process provides an interface through system management for the following operator-requested operations.

- Autoconfiguring devices
- Adding a counterpart to a shadow set
- Removing a counterpart from a shadow set
- Managing spare disks
- Disabling and enabling processors

The configuration-manager process keeps current configuration information on the system to enable it to reconfigure in the case of device failures or imminent failures. It does this by configuring the system at boot time an by monitoring the mounting and dismounting of Files–11 volumes, shadow sets, and stripe sets. Also, it keeps a current list of spare disks. A spare disk may be specified to be a spare for any disk of a compatible type, for a disk on a particular controller, or for a specific disk.

When notified of significant configuration error events, the configuration-manager process implements a strategy that maximizes the availability of the system. In this strategy the process performs the following operations:

- If a disk is failing, and if a spare exists, calls the shadow function processor to add the spare as a counterpart and to remove the failing counterpart.
- If a failing controller has a backup, initiates failover to the backup controller by calling the proper function processors.
- Uses the preceding two strategies to respond to a failing bus. In this case, if controllers on the bus have a backup, the configuration manager process makes the appropriate calls to failover to the backup controller. If there is no backup for a particular controller, the configuration manager process attempts to remove any disk attached to the failing controller from any shadow set of which it is a member and replace the disk with a spare disk attached through a controller on a good bus.
- Reports failing memory pages to memory management, which must—if possible—replace them and put them on the bad page list.
- Calls the configuration function processor to perform self-test and standalone diagnostics on failing processors. \Currently, the design has not considered automatically invoking diagnostics on disks or controllers - is this something to be included in the design?\
- Configures processors into the system or out of it by calling the configuration function processor.

34.1.2.2.3 Configuration Function Processor

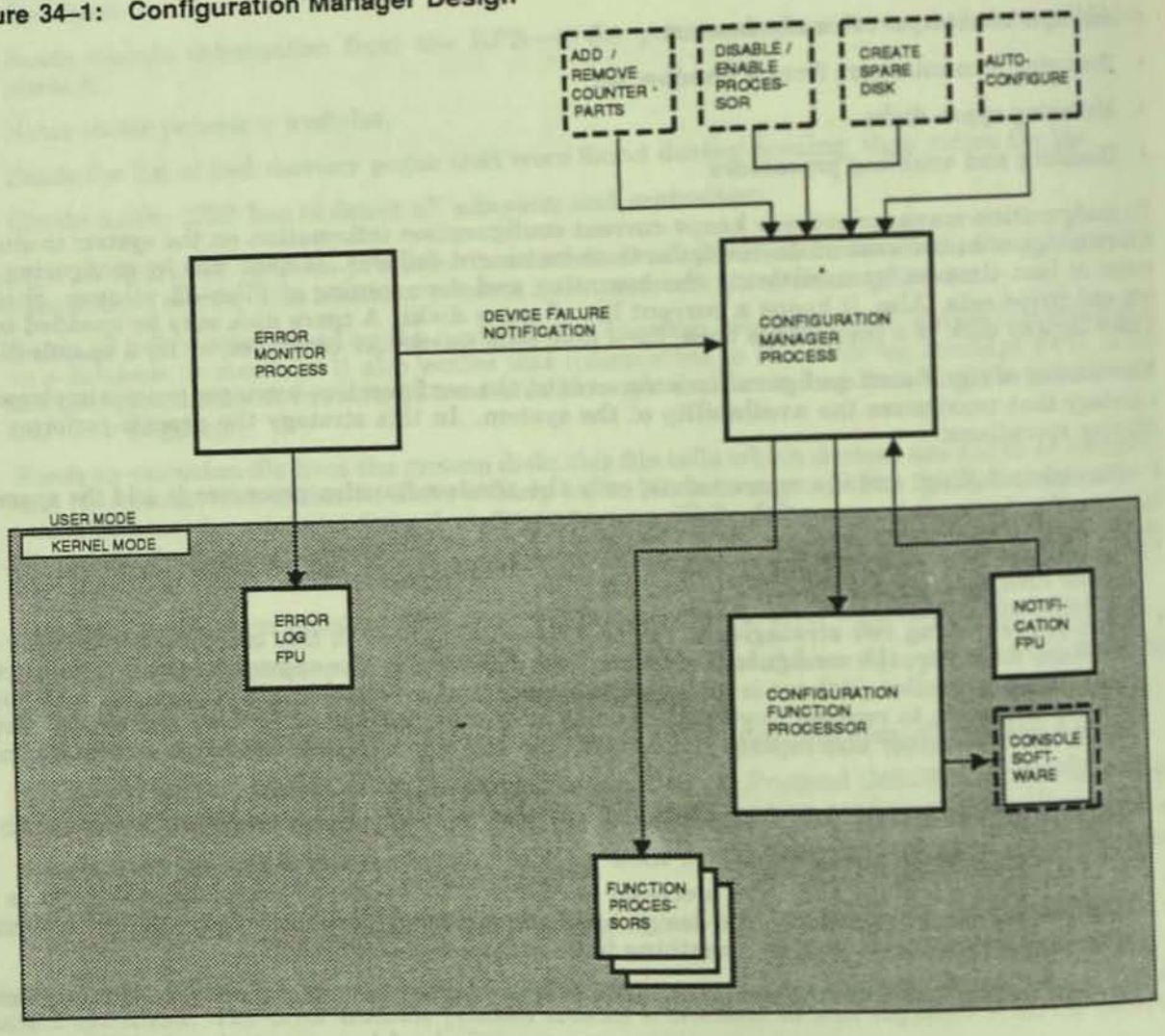
The configuration function processor performs the following tasks during normal system operation.

- Finds any new adapters and controllers in an online system. The configuration manager process commands the function processor to autoconfigure—that is, to find any new devices by checking the buses. The function processor then makes a list of new controllers or adapters, and returns this list to the configuration-manager process.
- Calls the console software interface in response to commands to initiate self-test and standalone diagnostics on processors. Returns results of the diagnostics to the configuration-manager process. This interface is described in the Chapter 20, Console Support.
- Calls the kernel software, the console software, or both in response to commands to disable or enable a processor.

34.1.3 Design

Figure 34–1 shows how the error-monitor process, the configuration-manager process and the configuration function processor interact.

Figure 34–1: Configuration Manager Design



34.1.3.1 The Error-Monitor Process

The error-monitor process reads error events logged to the error-log FPU, and predicts failures based on the information read. This user-mode process contains thresholds and algorithms that cause it to notify the configuration-manager process, indicating that a device has entered a state that may require action.

The monitor process would report as an example, the following events to the Configuration-Manager process.

- A processor has been disabled by the machine check code due to a non-recoverable error occurring.
- A disk has had an excessive (compared to threshold information kept by the monitor process) number of errors of a certain type.

- A disk has incurred a non recoverable error.

34.1.3.2 The Configuration-Manager Process

The configuration-manager process decides what strategy to follow when the error-monitor process notifies it that a device is failing or has failed. It then implements that strategy.

Depending on the problem, the manager process takes the appropriate action by calling device function processors to handle device errors, calling the configuration function processor to handle processor failures, or calling memory management to report bad pages.

If the device is one on which diagnostics can be run online, the decision is made after diagnostics are run and the results known to the configuration-manager. Then the decision is to attempt reconfiguration if *a*) the device fails diagnostic tests, or *b*) the device passes diagnostic tests, but has done so an excessive number of times after experiencing problems. If the device does not support online diagnostics, the reconfiguration is always attempted.

Whether a device can be configured out of the system or not depends on if there is a backup for that device or processor. For instance, if a controller has a backup, the manager process calls the MSCP function processor and the controller function processor to cause failover to occur. If a failing disk is part of a shadow set, it is removed from the set. If there is a spare disk which is an appropriate replacement, it is added to the shadow set. If there is no backup, the decision depends on the particular device and the type of error. In some cases, the only action that can be taken is to alert the operator, for instance in the case of a failing disk which is not shadowed; in other cases, the only action is to shut down the device, for instance, a fatal disk error.

The configuration-manager process handles user requests to enable or disable processors, to add or remove counterparts from a shadow set, or to autoconfigure a new device on a running system.

The shadow function processor provides an interface which is called by the configuration manager to request that a counterpart be added to a shadow set or removed from it. When calling this interface, the configuration-manager process requests the addition, and waits for the completion of the catchup before removing a failing counterpart. The configuration change is reported to the error-log FPU. Thus, the error log will contain a history of these configuration changes.

When the configuration-manager process calls lower-level function processors to cause failover to a backup controller, these changes are recorded in the error log.

The console software provides an interface to enable and disable processors. The configuration manager process makes the decisions to perform these actions either

- In response to a user request
- To reenable a processor which was disabled by machine check as the result of a failure, but which passes diagnostics such that the configuration manager decides it can continue to be used

These processor changes are sent to the error-log.

The configuration-manager process keeps an internal database that represents the current configuration. This database contains information about buses, controllers, adapters, disks, tapes, Files–11 volumes, shadow sets, and stripe sets. This information is used to respond to failure situations.

The configuration-manager process mounts disks at system boot time, doing so in the following manner. As the MSCP function processor finds disks, it writes information about them to the notification FPU. The configuration-manager process reads this information from the notification FPU, then issues I/O requests to MSCP to ready the disk, unless the disk is in the exclusion file. Next, the configuration-manager process requests that the MSCP function processor read the context area of the disk. This area describes how the disk is to be mounted, including any stripe sets or shadow sets to which the disk belongs. This area is maintained by the function processors which do the mounting. The configuration-manager process then mounts the disks based on the information in the context area. Parallelism is introduced to minimize the time needed to accomplish the mounting by creating multiple threads to do the mounting.

If the system has an external service processor running SPEAR or other SDD tools, the Error Monitor process can receive additional problem notification from this external processor. This interface is TBD.

34.1.3.3 The Configuration Function Processor

To find devices that are to be configured into the system, the configuration function processor checks the buses. It does so by looking in I/O space at the possible node locations, and reading the device register. This information is returned to the configuration-manager process through an I/O request.

The configuration-manager process requests that the configuration function processor invoke self-test diagnostics and standalone diagnostics on a processor if either *a*) the number of recoverable errors on that processor exceeds a threshold, or *b*) the processor is hung or has encountered a nonrecoverable error. The configuration function processor returns the result of the diagnostics to the configuration-manager process.

The configuration function processor interfaces with the console software to initiate diagnostic tests and to get the results of such tests. The configuration-manager process logs the diagnostic results to the error-log FPU.

The configuration function processor does hardware disables and enables of processors by calling a console software library routine. Processors are software disabled and enabled by calling the kernel.

34.1.4 Relation to Other Software

The configuration management software is, in most cases, only an agent of reconfiguration. Often, it reports failing hardware (for example, failing memory) to other software, or it makes requests of other software when it detects imminent failure. Generally, the configuration-manager process decides only that a particular device should be configured out of the system, if possible. The process then calls other software to configure out the device. This section describes the other Mica components which are involved in configuration management.

34.1.4.1 Memory Management

The configuration function processor calls memory management to report a failing page. If possible, memory management puts the page on the bad page list.

34.1.4.2 Shadow Function Processor

When called to replace a failing disk with a spare counterpart. The shadow function processor updates the spare disk. Also, the shadow function processor is called to remove the failing counterpart after the replacement is up to date.

34.1.4.3 MSCP Function Processor and Controller Function Processor

The MSCP function processor releases a failing controller; the controller function processor unreadies one.

34.1.4.4 SCS Function Processor

The Configuration Manager process calls the SCS function processor at system startup with the list of FPUs it has created for the adapters and controllers found on the bus.

34.1.4.5 Device Function Processors

When the Configuration Manager process performs autoconfiguration of a device, it calls the appropriate device function processor to create an FPU for any controller it finds on a bus (if the FPU has not been previously created).

34.1.4.6 Machine Check

The design described in this overview assumes that, in a multiprocessor environment, machine check does the following tasks for processor errors:

- Logs the error to the error-log FPU, recording the reason for the error.
- Decides whether to crash or not. If not, machine check signals the user thread that was executing when the machine check occurred.
- Disables the processor in all cases, whether the error was recoverable, nonrecoverable, or the processor was hung. The configuration-manager process may reenable the processor, based on thresholds and on the results of diagnostic tests, if it decides to run them.

34.1.4.7 Console Software

The console software provides an interface to request that processors be hardware disabled or hardware enabled. Also, it provides an interface to initiate self-test and standalone diagnostics on processors. Finally, it provides an interface to disable memory modules.

34.1.4.8 External Service Processor

The ESP will also inform the Error Monitor of imminent hardware failures. This is based on the assumption that the proprietary software running on the ESP will be more sophisticated in its ability to predict imminent hardware failures and will provide an additional level of availability above that provided by the Error Monitor's simpler thresholding.

The Configuration Manager software does not send any information to the ESP.

This interface from the ESP to the Configuration Manager process is TBD.

34.1.5 Issues

The following issues are unresolved:

1. We need to formalize the interaction of machine check and the configuration manager software.
2. Either the Configuration Manager can log configuration changes or the function processor that implements the change can do this. If the Configuration Manager logs the changes, the function processors must protect the action such that only the Configuration Manager can request the changes.
3. We need to define the interaction with Network software—DECnet and IPC for Cheyenne.

CHAPTER 35

SYSTEM VOLUME LAYOUT AND SOFTWARE INSTALLATION

35.1 Overview

This chapter describes the system volume layout for Mica operating system directories and files. The chapter also describes software installations and updates for the operating system, layered products, and third-party and client software. The information provided in this chapter applies to both the Cheyenne and the Glacier products.

35.1.1 System Volume Layout

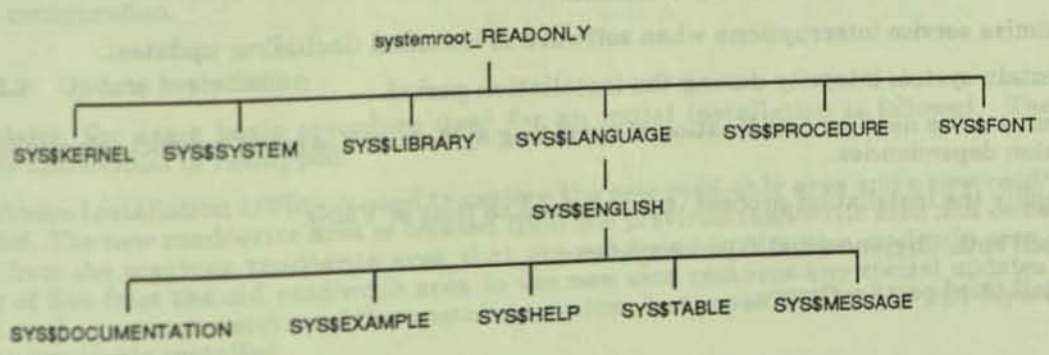
The system volume is divided into two areas, a read-only area and a read/write area. The directories in the read/write area are a superset of the directories in the read-only area. The following two sections describe these areas.

35.1.1.1 The Read-Only Area

Figure 35–1 shows the directories contained in the read-only area.

The read-only area is created initially by the Mica operating system installation. After the installation, no modification is done to the read-only area, which eliminates the need for backups of this area. Most operating system modules reside in this area.

Figure 35–1: Read-Only System Volume Area

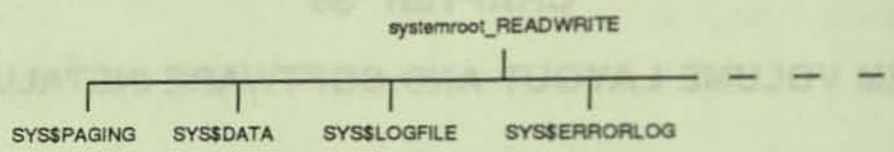


35.1.1.2 The Read/Write Area

The read/write area includes those directories shown above in the read-only area plus the additional directories shown in Figure 35–2.

The read/write area is also created by the Mica operating system installation. However, modifications to this area occur during the life of the system. Such things as paging files, error logs, and configuration files are stored in the read/write area directories. In addition, layered products, special operating system updates, and field test updates are also placed in the directories in this area. This area requires backups.

Figure 35–2: Read/Write System Volume Area



35.1.1.3 Read-Only and Read/Write Area Interaction

Because the root directories for both the read-only area and the read/write area are referenced through a single search list logical, SYS$SYSROOT, these areas appear as a single area. The read/write area is referenced first in the search list so that files placed in the read/write area can supersede those in the read-only area.

For booting purposes, the console subsystem maintains a pointer to the read/write area. In turn, a known file in the read/write area contains a pointer to the read-only area. See Chapter 12, Booting, for more details.

35.1.2 Software Installation

There are three types of Mica installation procedures: standard, special, and front-end/client. The first section below lists the overall goals of these installation procedures. The remaining sections describe the Mica installation strategy in general, the three types of installation procedures, and considerations for high-availability configurations.

35.1.2.1 Goals

The following are the goals of Mica installation:

- Minimize service interruptions when software is installed (including updates).
- Maintain system integrity during the installation period.
- Eliminate the need for coordination of operating system and layered product releases because of version dependencies.
- Simplify the installation process (as compared to that of VMS).
- Install both Cheyenne and Glacier systems.
- Install third-party software.

35.1.2.2 General Description

The software distribution strategy of the Mica operating system and layered products is that, for the general case, there is no installation procedure *per se*. Instead, both initial systems and updates are *pre-installed* and *complete*. *Pre-installed* means that after the system has been transferred from the distribution media to the system disk, the system is ready to run. *Complete* means that the distribution contains the full operating system and layered products. This type of installation is known as a standard installation. The standard installation is the general procedure used for initial system installations and updates.

Since all products are shipped to the customer, the License Management Facility (LMF) is used to provide access to optional products purchased from DIGITAL. Software releases are coordinated among the operating system and layered products. Simplification of the release process is done by scheduling releases at regular intervals, such as quarterly or semiannually.

The standard installation procedure is not practical for third-party software, field test updates, and emergency module replacements (patches). To accommodate this software, an installation procedure similar to VMSINSTAL is provided. This procedure is known as a special installation. This is the procedure used for software that cannot be included in the standard distribution; for example, third-party applications and new layered products.

35.1.2.3 Standard Installation

The major technical design that makes the standard installation strategy work is the layout of the system disk. The device used for the system read-only area need not be a read-only device. However, maintaining strict rules as to the use of the read-only area allows future hardware configurations to incorporate such devices without software modification. Assuming the device is a read/write disk, the disk may be removable or nonremovable. For FRS, it is assumed that the disk will be nonremovable.

There are two types of standard installations: initial installation and update installation.

35.1.2.3.1 Initial Installation

An image backup of the read-only area is shipped to the customer, who then restores it to the system disk using the Software Installation Utility.

After the read-only area has been restored, the read/write area is created and populated with the default configuration.

35.1.2.3.2 Update Installation

For updates, the same basic procedure used for an initial installation is followed. The complete software distribution is reshipped.

The Software Installation Utility is used to restore the new read-only area and a new read/write area is created. The new read/write area is created from the previous read/write area and contains copies of files from the previous read/write area that are not found in the new read-only area. Selective copying of files from the old read/write area to the new area removes any special updates that were applied to the previous version of the operating system, but saves third-party and layered product software previously installed.

35.1.2.4 Special Installation

Special installations apply to third-party software, layered products not released with the operating system, field test updates, and emergency module replacements.

35.1.2.4.1 Special Installation Types

There are three possible types of special installation:

- *Off-line* refers to installing software after user activity on the system is stopped and disabled. After the installation is complete, user activity is enabled, and applications are started, using the new underlying software. The current VMS installation procedure is this type of installation.
- *On-line* refers to installing software while the system is running user applications. For on-line installations, the application must be restarted to use the new version of the underlying software.
- *Hot module replacement* refers to installing software while the system is running user applications and then using the new underlying software without affecting the application.

On-line installation procedures can be developed for the general case. Hot module replacements must be developed depending on individual application requirements and their interaction with the underlying software. General purpose hot module replacement is not in the scope of Mica software installation.

Special installations on Mica affect the read/write area of the system disk and look much like VMSINSTAL, with the exception that the procedure supports on-line software installation.

35.1.2.4.2 Special Installation Procedure

The special installation procedure installs *n* number of products, updates, etc., and causes them to appear on the system simultaneously. The Software Installation Utility performs the following steps for special installations:

1. Creates a temporary, inactive read/write area on the same disk as the current active read/write area.
2. Populates the temporary read/write area with the product/update. Any files that require modification are copied from the read/write or read-only areas into the temporary read/write area and then modified.
3. Causes one of the following two sequences to occur.
   a. If the incorporation of a product or update requires the operating system to be rebooted:
      i The system reboot sequence is initiated. During shutdown, after all user activity has been terminated, all of the files in the temporary read/write area are renamed into the active read/write area.
      ii The temporary read/write area is deleted.
      iii The system shutdown continues and the system reboots.
   b. If the incorporation of a product or update does not require the operating system to be rebooted:
      i The system search list of "read/write area, read-only area" is changed to "temporary read/write area, read/write area, read-only area". The new products and updates appear on the system at this time.
      ii Files in the temporary read/write area are renamed to the real read/write area.
      iii The system search list is restored to "read/write area, read-only area".
      iv The temporary read/write area is deleted.

35.1.2.5 Front-End and Client Software Installation

VMS front-end and client software is installed according to the VMS guidelines, using VMSINSTAL. High availability could become an issue for the overall system (client through database machine), since VMS installation procedures do not provide the same availability as do Mica installation procedures.

Ultrix client software is installed according to Ultrix guidelines, using the *setld* command.

Client software is shipped on media separate from the server software and meets client media requirements.

35.1.2.6 High-Availability Configuration

For the Cheyenne/multiple Stone configuration, one Stone system is upgraded at a time. During the installation process, only one Stone system is rebooted at a time. This causes the system workload to be moved from one Stone system to the others while it reboots the new version of software. The workload balances out over the database server as the last Stone system is rebooted. The database server is therefore available 100% of the time during the upgrade.

This procedure does not eliminate the need for compatibility between Mica version n and Mica version $n + 1$ on the Cheyenne/multiple Stone configuration. After one machine is upgraded and before the other machines are upgraded, version n and $n + 1$ will be running on the same database server.

Testing and Performance Measurement

This set of chapters describes testing and performance measurement on Mica.

CHAPTER 36

PERFORMANCE MONITOR

36.1 Overview

This chapter describes the *Monitor* utility for the Mica operating system. This utility displays and records information about system resource usage on a Glacier or Cheyenne system.

Mica Monitor is available only from a VAX/VMS client for FRS. Our long-term goal is to run on all supported clients.

36.1.1 Goals

Mica Monitor is designed to achieve the following goals:

- Provide a tool that displays, records and summarizes system performance data for a "live" system
- Provide usage data for the widest possible range of Mica system resources
- Use a minimum of system resources to gather the performance data
- Enhance maintainability and reliability of the code by using well-defined interfaces to gather performance data
- Maximize flexibility so that the Mica Monitor utility may be extended in the future

36.1.2 Terminology

In discussing the Mica Monitor utility, the terms *class*, *level*, and *rate* mean the following:

- Class—A group of data items that provide a statistical measure of the performance of a particular subsystem
- Level—The current value of a data item, that is, a "snapshot"
- Rate—The number of occurrences per second

36.1.3 Functional Overview

Mica Monitor collects data on systemwide usage of Mica resources at user-specified sample intervals from a Glacier or Cheyenne system. Data is organized into classes. The user specifies the classes of information to be collected.

The raw data can be recorded to a binary file. Either current or previously recorded data can be processed and displayed to a user screen at user-specified viewing intervals. A summary of the data can also be computed and written to an ASCII listing file.

Each data item is defined as a rate or a level. Current, as well as simple average, minimum, and maximum values over the duration of the MONITOR request are calculated for each item, and displayed to the screen. In general, one screen image is produced per requested class per requested viewing interval. The data items for a particular class may require more than one screen.

36.1.3.1 Time Intervals

The user may specify the *observation period*, *sample interval*, and *viewing interval*:

- Observation period—The beginning and ending times for viewing or summarizing current or previously recorded data
- Sample interval—The time interval at which systemwide performance data is to be collected and computed
- Viewing interval—The time interval at which current or previously recorded data is to be displayed to the user screen

The minimum sample and viewing intervals defined are one second.

36.1.3.2 Classes

Classes of information to be collected for Mica include the following. This is not intended to be a complete list.

- Modes
- Thread states
- Page faults
- Disk I/O
- File system caching
- DECnet
- System summary

The Mica Monitor chapter will include a specification of commands supported.

36.1.4 Implementation Overview

The Mica Monitor server gathers operating system data by calling the kernel mode system service *exec$get_system_performance*. This system service is described in the Internal System Services Manual.

The bulk of data manipulation, recording, and display is performed by a user-mode program which runs on the client VAX/VMS system. To gather data, the client portion initializes a server on the Mica/Stone system, then calls for data from the server at the requested sample interval until either:

- the requested observation period has ended, or
- the user issues an exit request

The client interface to the server is via RPC.

The user interface to Mica Monitor is part of the Glacier or Cheyenne System Management interface. For character cell terminals, a minimal input and display interface is built on the VAX/VMS Screen Management Run-Time Library (SMG). Mica Monitor may also be invoked via a command file.

A DECwindows interface is provided for bit-mapped devices. The displays provided for the DECwindows interface are more graphically oriented than those for terminals.

If summarizing is requested, the summary output is written to a listing file.

Mica Monitor provides a recording capability. Data is recorded to a binary data file and then played back to a user screen or summarized through the Mica Monitor utility. The format of the recording file will not be documented.

36.1.5 Issues

Mica Monitor requires a CLI interface. The interface for character cell terminals will be based on SMG. Do we provide a separate interface for DECwindows devices for FRS, or punt with terminal emulation for those devices?

Monitor will follow the same interface strategy as system management, in order to provide a consistent user interface.

CHAPTER 37

USER-LEVEL SYSTEM EXERCISER

37.1 Overview

The *User-Level System Exerciser* (USE) is similar to the VAX/VMS User Environment Test Package (UETP). It enables Digital manufacturing and field service to exercise Glacier on the manufacturing floor and at the customer site. *USE* also functions as an installation verification procedure (IVP) that can be run at system installation to ensure that the hardware and software have been properly installed.

USE simulates the use of a Mica system on a single Moraine or Stone box. USE tries to exercise the Moraine or Stone hardware and devices connected to it. USE also exercises many of the functions in the Mica software.

37.1.1 Goals

The goals for USE are:

- To perform the traditional UETP tasks of testing the hardware connections to peripherals and of ensuring that the operating system has been correctly installed
- To function as a user-level systems exerciser
- To subject the system to a load test in order to stress the system
- To provide the user with an interface that fits in the spectrum of diagnostics
- To exercise aspects of the system that are specific or critical to Glacier

USE tests tape drives, disk drives, the ethernet, the CPU, and memory. It does not perform any console or terminal testing.

37.1.2 Non-Goals

USE does not:

- Test every aspect of the Mica operating system or every feature of the Moraine or Stone hardware exhaustively
- Test any layered product, such as a compiler
- Exercise or test any client front end; no hardware or software problems specific to the clients will be detected by USE—this can be done by running the client system UETP
- Test or utilize the Quartz software
- Exercise multiple Moraine or Stone boxes and their front ends concurrently

37.1.3 Outline of the Functionality

USE is invoked from the client system or console through the PRISM Diagnostic Monitor (PDM) and executes exclusively on a Moraine or Stone running Mica. Any information that needs to be presented to the user is transmitted to the client through PDM. See Figure 37–1.

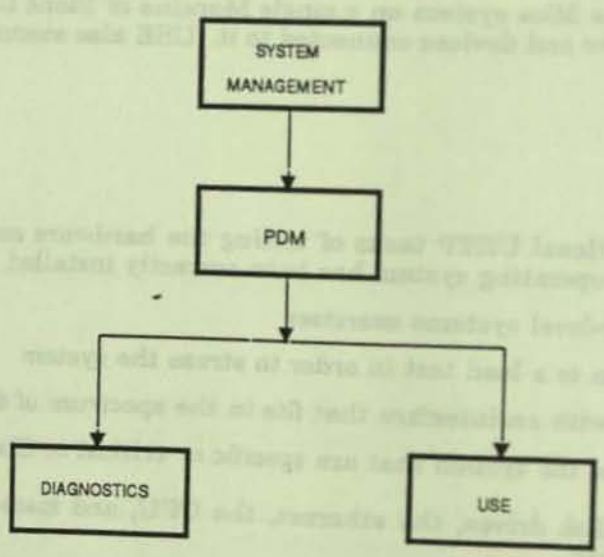
USE performs individual tests of the devices that the user has selected for testing. It also tests the Moraine or Stone hardware and all its devices in unison, simulating a multiprogramming environment. USE attempts to create a heavy load on the system and to test Glacier in different ways.

USE provides a moderate level of isolation capability. For example, it can tell the user which device or device controller failed when that particular device's test fails. However, there are instances when errors will occur for which USE will have either no or imprecise isolation information.

Subsets of USE can be run so that the user can focus testing on particular aspects of the system. These subsets consist of the individual device tests or the load tests.

When there are no errors, USE displays a message on the terminal saying that the system passed USE testing. Errors are reported as they occur. In addition, as USE executes each of the tests, any errors along with information about the failing test and device are entered into a log file through PDM.

Figure 37–1: Interface Hierarchy of USE



37.1.3.1 Interactions with Other Software

USE, although using the same interface, does not directly interact with the diagnostics. It does not use them or depend on them.

The isolation capabilities of USE depend on the Error Logger and Symptom Directed Diagnosis (SDD). As errors occur during the execution of USE, they are logged by the Error Logger and diagnostics are invoked by SDD to deal with specific problems.

USE executes exclusively on the Moraine or Stone system. Therefore, USE cannot test any of the functionality of the front ends, the software that executes there, or the configuration of multiple Moraine or Stone boxes and client systems. This type of testing can be done by systems exercisers developed for the client systems.

USE depends on PDM to perform all the communications with the client system.

37.1.4 Outline of the Design

USE consists of three main sections:

- Input and Initialization
- Device Testing
- Load and Application Specific Testing (LAST)

37.1.4.1 Input and Initialization

USE obtains inputs from the user through PDM. Other system parameters are obtained at this time—such as the number of CPU modules, the amout of memory, and so on. All of this information is obtained through Mica operating system services. This information is used to calculate parameters about the volume of loads and the types of tests to be performed.

37.1.4.2 Device Testing

The basic idea behind device testing is to write a specific, known data pattern to the device and make sure that it is done properly. In the case of testing disks or tape drives, the written data can be re-read to ensure that it was written correctly. For the ethernet, loopback mode is used to verify the transmitted packets. If a device cannot be accessed, then testing of that device is aborted and no effort will be made to test it in the subsequent load testing.

37.1.4.3 Load and Application Specific Testing

The load test creates a large number of processes, depending on the CPU configuration and other resource parameters. The number of processes and even the tasks they are to perform would increase with the number of processors. These processes include some device tests as well as user-level code. This user code is chosen so as to exercise the CPU, memory, and the devices in the way that users would most likely utilize the system. This specialized testing is now described in greater detail.

37.1.4.3.1 Testing Glacier

Computation-intensive Pillar code is chosen to stress and to exercise the system and can be run on the system directly. This is discussed in Section 37.1.5.

Hardware errors are logged by the Error Logger, which functions independently of USE. Errors that are visible to USE are reported to the user and, if not fatal, testing continues.

37.1.4.3.2 Fault Tolerant Testing

Fault tolerance specific to a Moraine or Stone box is tested during the load test. As the load or stress test is running on a Cheyenne system, a process running on the same system asynchronously removes processors from the configuration at random intervals. Disks can be removed from the configuration through a slightly different technique. This macroscopic level of fault tolerance checking is all that USE does.

37.1.5 Developing Glacier User Tests

USE utilizes typical or representative user-level tests for Glacier.

Tests for Glacier can consist of industry-standard benchmarks, such as the Whetstone or Linpack benchmarks. These are available in FORTRAN. Less complex tasks, however, would probably be sufficient. Inversion of large matrices typical of those that occur in the numerical solution of partial differential equations or finite element analysis would exercise the parallel processing capabilities of Glacier.

37.1.6 Requirements

USE does not place formal requirements on the hardware or the operating system, since it is the job of USE to test a designed system as a user would probably utilize it, not to constrain the design. However, there are certain areas of support which would facilitate the flexibility and usefulness of USE.

37.1.6.1 User Diagnostics Interface

PDM is the means by which the user will invoke USE. PDM needs to be completed before USE can be completed.

37.1.6.2 Error Logging and Symptom Directed Diagnostics

USE does not perform any extraordinary isolation of faults. It does, however, identify devices that generate errors or whose tests have otherwise failed to perform as expected. It does not identify faults down to failing FRUs. Many errors reported by USE could be caused by a variety of disparate factors. The Error Logger coupled with SDD is expected to identify points of error, invoke the diagnostics, and warn the user. In this way, a user can obtain isolation information and close the information gap between USE and the diagnostics.

37.1.6.3 Mica System Services

USE requires a means to logically remove processors and disks from the Moraine or Stone configuration in order to simulate macro-level faults.

37.1.7 Open Issues

Open issues that must be settled are:

- What are the specific user-level applications tests for Glacier? How many are required?
- What are the algorithms to determine the volume of load on a system?
- What are the means to logically remove processors or disks from Moraine or Stone?

Network

This set of chapters describes the network-related components of Mica.

CHAPTER 38

MICA NETWORK OVERVIEW

38.1 Overview

The Mica-based products (Glacier and Cheyenne) are designed to work only in networked environments. In this sense, Mica is unique among systems of its size. Mica relies heavily on communications; therefore, all parts of its network implementation directly affect the success of some portion of the system.

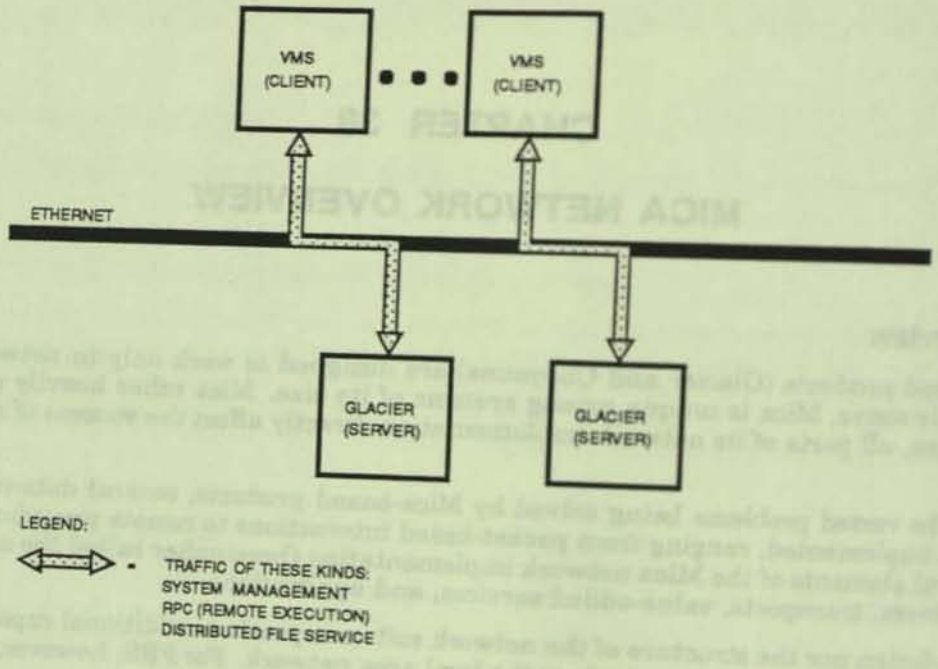
Because of the varied problems being solved by Mica-based products, several data-communications methods are implemented, ranging from packet-based interactions to remote procedure calls (RPCs). The structural elements of the Mica network implementation (hereinafter called the *network*) include data-link drivers, transports, value-added services, and applications.

Neither the design nor the structure of the network software preclude additional capabilities, such as directly connecting interactive terminals to the local area network. For FRS, however, the Mica-based products do not require such capabilities.

This chapter is not intended as a technical reference for Mica networks; such technical detail is covered in other chapters. Instead, this chapter presents the high-level requirements and goals for Mica networks, and describes how these goals are satisfied.

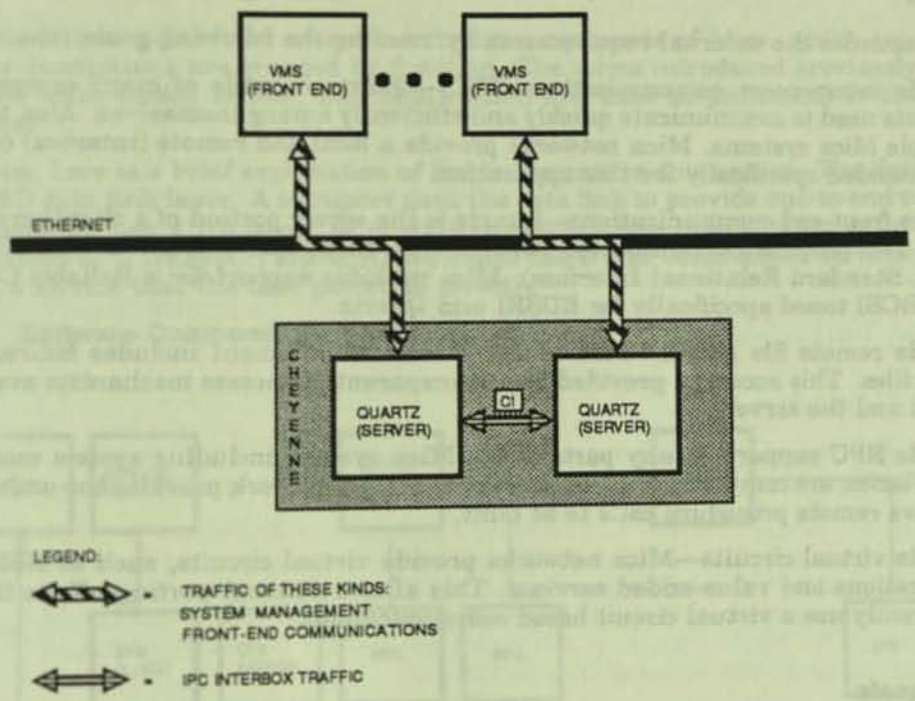
Figure 38–1 and Figure 38–2 show the relationships among Mica-based systems and clients of these systems. These figures show simple and typical network topologies along with the type of communications traffic among the systems.

Figure 38–1: Glacier Communications



There are three primary uses of the network to support Glacier servers. Glacier servers use Remote Procedure Calls (RPC) to effect remote execution of work for clients. A description of the use of RPC for Glacier support can be found in Chapter 50, Glacier Overview. The client and server use the Distributed File Service (DFS) for remote access to each others disk based files. DFS is documented in Chapter 43, Distributed File Service Introduction. Management clients communicate with Glacier servers through system management requests. System management's actual communication (described in Chapter 32, System Management) is layered on RPC.

Figure 38–2: Cheyenne Communications



The client of Cheyenne is called a *front end*. Multiple Mica systems running Quartz software can make up a single Cheyenne. The traffic between a front end and a Cheyenne can be either database transactions (using front end communications) or system management requests. There is also traffic between individual systems in a Cheyenne for managing distributed databases and server failover.

38.1.1 Requirements

The network meets the following external requirements:

- Support of Cheyenne—The network must provide interbox and Quartz front-end communications that can sustain the goals for Cheyenne performance, availablility, and reliability (see Chapter 48, Cheyenne Overview).
- Support of Glacier—The network must provide the base for Glacier client-to-server communications and for remote access to disk based files.
- Support of Mica system management—The network must provide a method for remote system-management clients to communicate with their respective servers on Mica.
- Heterogeneous interoperability—Mica must coexist with VAX/VMS and ULTRIX systems, and with other Mica systems. The network must provide the underlying communication mechanisms that enable these systems to interoperate in the context of Mica-based products.

38.1.2 Goals

The network satisfies the external requirements by meeting the following goals:

- To provide interprocess communications (IPC)—Quartz is made of many components. These components need to communicate quickly and efficiently among themselves. Also, they can reside on multiple Mica systems. Mica networks provide a local and remote (interbox) communication facility intended specifically for this application.
- To provide front-end communications—Quartz is the server portion of a client/server distributed system. The protocol communicated between the client and the server is EDSRI (Extended DIGITAL Standard Relational Interface). Mica provides support for a Reliable Communication Service (RCS) tuned specifically for EDSRI and Quartz.
- To provide remote file access—Part of the Glacier environment includes bidirectional remote access to files. This access is provided by a transparent file-access mechanism available on both the client and the server.
- To provide RPC support—Many parts of the Mica system, including system management and most of Glacier, are composed of client/server pairs. The network provides the underlying support that allows remote procedure calls to be built.
- To provide virtual circuits—Mica networks provide virtual circuits, such as NSP logical links, for applications and value-added services. This allows minimal porting efforts for applications that currently use a virtual circuit based communications.

38.1.3 Nongoals

The following items are not among the goals that must be met for the network to satisfy the external requirements.

- To provide a complete implementation of DNA * Phase V at FRS—Protocols such as TP4 will be implemented after FRS. Routing will be restricted, initially, to be end node only. Protocols such as DAP and CTERM will, possibly, never be implemented.
- To provide a comprehensive network environment that isolates users from locality or knowledge of the existence of the network—This transparency is the function of higher layers of software. (For details, see Chapter 50, Glacier Overview.)
- To support VAXclusters or any other common-management-domain groups, such as workgroups.
- To solve the problem of distributed security—The design of the Mica security system (see Chapter 10, Security and Privileges) is flexible enough to adapt to a secure network, when the architecture becomes available.
- To directly connect to wide area networks—The primary environment that Mica-based systems will reside is on a local area network. Wide area networks, although implicit in DNA, are not directly connected to Mica systems.
- To support DNA on the CI—It is not currently possible to use the CI effectively with DECnet from a VAX/VMS system. Since VMS systems are the major systems on a CI that a Glacier system would wish to communicate, DECnet CI support will not be provided until the VMS performance is improved.

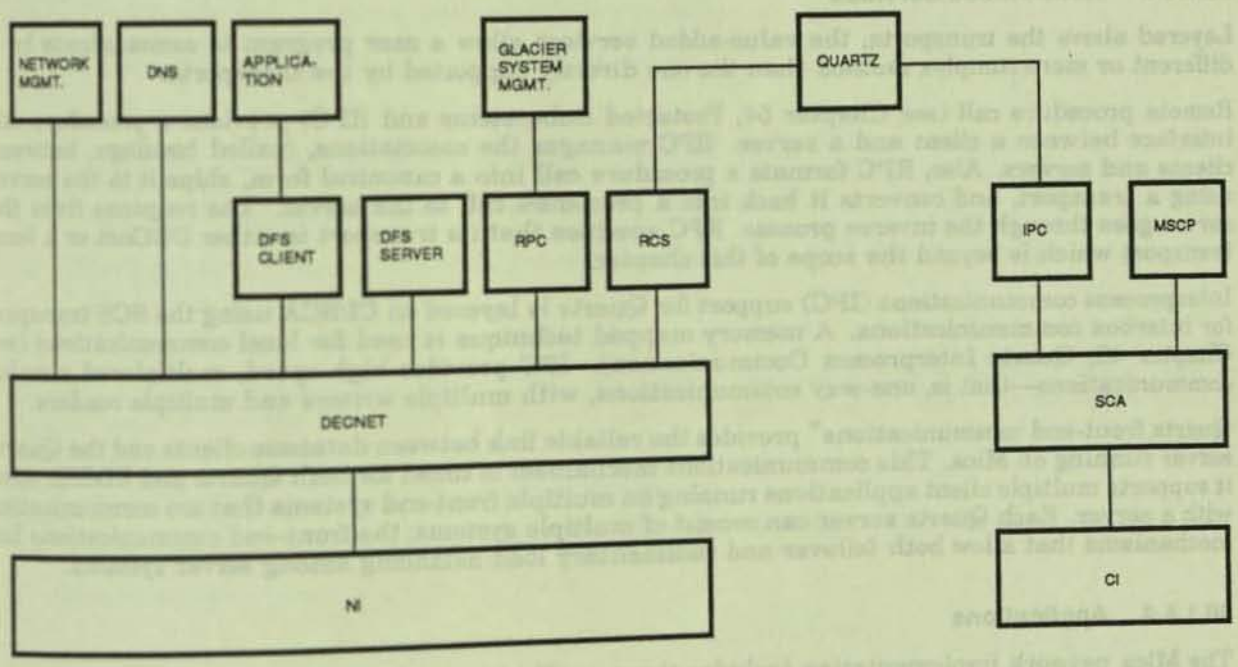
* DIGITAL Network Architecture

38.1.4 Network Software Components

This section describes each of the network software components of Mica, which are shown in Figure 38–3. The descriptions are grouped by function. The terms introduced previously (such as *data link*, *transport*, *value-added service*, and *application*) are used in preference to those of the ISO reference model.

For convenience, here is a brief explanation of the terms used in this section. The *data link* is equivalent to the ISO data link layer. A *transport* uses the data link to provide end-to-end communication. *Value-added* services use a transport to provide communication that is different or more complex than that provided by a transport alone. *Applications* use either value-added services or a transport to implement a service that the user perceives directly.

Figure 38–3: Software Components of the Network



38.1.4.1 Data Link

The NI (Network Interconnect) and the CI (Computer Interconnect) are the supported communications media for Mica. Both are local area networks. Mica does not directly support wide-area networks.

The NI is used by the DNA transports. Phase V of DNA uses the NI as a data link for an Ethernet or IEEE 802.2 network. The NI transmits both to single and multicast addresses.

The CI is used by the SCS transport.

For both kinds of data links, a Mica system typically includes multiple controllers. To increase availability and performance, Mica systems are sometimes connected through multiple paths. The higher layers, such as the transports, are responsible for effecting these gains. The data links connect the system to the immediate local network only.

38.1.4.2 Transports

SCS is used for tape and disk access (see Chapter 17, System Communication Services) and IPC (see Chapter 42, Quartz Interprocess Communication. If multirail use of the CI is desired, it is the responsibility of the users (SYSAPs) of SCS to figure how they want to do it. SCS provides an interface only to kernel-mode users.

The DECnet transport is NSP (Chapter 39, Network Services). Between NSP and the data link, DECnet provides a network services layer that performs routing. Routing supports end node functions only (that is, it does not provide packet forwarding services). This layer allows connectivity with systems not on the same immediate local area network as the Mica system. Also, the network services layer supports multirail access for increased performance and availability. Since the DECnet transport is the primary communications mechanism for Mica, its implementation is highly tuned for performance. DECnet provides both a kernel-mode and a user-mode session interface.

38.1.4.3 Value-Added Services

Layered above the transports, the value-added services allow a user program to communicate by a different or more complex method than the one directly supported by the transports.

Remote procedure call (see Chapter 54, Protected Subsystems and RPC) provides a procedure-call interface between a client and a server. RPC manages the associations, (called *bindings*, between clients and servers. Also, RPC formats a procedure call into a canonical form, ships it to the server using a transport, and converts it back into a procedure call to the server. The response from the server goes through the inverse process. RPC specifies that its transport is either DECnet or a local transport which is beyond the scope of this chapter.

Interprocess communications (IPC) support for Quartz is layered on CI/SCA using the SCS transport for interbox commmunications. A memory mapped technique is used for local communications (see Chapter 42, Quartz Interprocess Communication). IPC provides high-speed, multiplexed simplex communications—that is, one-way communications, with multiple writers and multiple readers.

Quartz front-end communications* provides the reliable link between database clients and the Quartz server running on Mica. This communications mechanism is tuned for both Quartz and EDSRI. Also, it supports multiple client applications running on multiple front-end systems that are communicating with a server. Each Quartz server can consist of multiple systems; the front-end communications has mechanisms that allow both failover and rudimentary load balancing among server systems.

38.1.4.4 Applications

The Mica network implementation includes three applications: distributed file service (DFS), DNA naming services (DNS), and DNA network management. The distributed file service (see Chapter 43, Distributed File Service Introduction) provides remote file access among VMS and Mica systems. The current implementation propagates the VMS XQP QIO interface through a request/response protocol layered on DECnet NSP.

DNA naming services (DNS) provides a tool for distributing and managing the global name space. DNS is used by the DNA session to translate node names into node addresses. Also, DNS is used by DFS to resolve volume-access points into the appropriate server node. Mica implements only the clerk portion of DNS(see Chapter 40, DNA Naming Service Clerk).

\DNA Time Services are not currently being implemented because we lack a time provider on VMS. DNS will just have to wing it and hope for the best with unsynchronized clocks.\

* Also known as Reliable Communications Services (RCS). It may revert to just Front-End Communications when the ongoing design discussions for this service are completed.

Mica provides DNA network management only in server form (see Chapter 41, DECnet Startup, Shutdown, Management, and Logging). We assume that DECnet–VMS NCL (management client) will be available to control the Mica network. On Mica, the only parts of the network implementation that are controlled by network management are those specified by the DIGITAL Network Architecture. All other components have their own management tools separately defined and implemented, usually in cooperation with the Mica system-management design group).

CHAPTER 39

NETWORK SERVICES

39.1 Overview

This chapter covers the following aspects of DECnet–Mica:

- The parts of Phase V that DECnet–Mica implements
- The user interface to DECnet–Mica
- The software components used to implement DECnet–Mica
- Unresolved issues regarding the DECnet–Mica project

Phase V is a corporate standard, so this paper does not address architectural issues.

39.1.1 Requirements and Goals

The selection of DECnet as a communications architecture is actually a design decision based upon the needs of several separate components. Following are the components whose requirements resulted in the selection of DECnet:

- Front-end to back-end Cheyenne communications
- Remote system-management communications in both Cheyenne and Glacier
- Client-to-server communications in Glacier

The requirements that resulted in selection of DECnet are as follows:

- Reliable virtual-circuit support
- Interoperability with the VMS, ULTRIX, and (possibly) MS-DOS™ operating systems
- Support for the DIGITAL corporate RPC

Like all DECnet implementations, DECnet–Mica must conform to the DIGITAL Network Architecture, and must include the minimum subset of modules defined by that architecture.

DECnet–Mica must support the NI as a data-link device.

Where possible, DECnet–Mica should be optimized for performance when supporting Cheyenne communications.

To help provide fault tolerance in Cheyenne, fault-tolerant features of DNA should be included in DECnet–Mica. Primary among these features is automatic circuit failover within the routing layer of Phase V.

™ MS-DOS is a trademark of the Microsoft Corporation.

39.1.2 DNA Components

DECnet–Mica is a small subset of Phase V, including only those components needed to implement a high-performance, fault-tolerant, end-node DECnet. That is, DECnet–Mica includes the following Phase V components:

- DNA naming services
- Network communication services
- Network management
- Network event-logging server

DNA Naming Services. In Phase V, both the session layer and the distributed file services (DFS) need the distributed name services provided by the DNA naming services (DNS). DECnet–Mica implements only a DNS clerk.

Network Communication Services. DECnet–Mica provides network communication services by implementing the Phase V specifications for the session layer, the Network Services Protocol (NSP), and the routing layer (end-node only). DECnet–Mica supports communications only over the the NI.

Network Management. For most network management tasks, DECnet–Mica nodes are controlled from a remote DECnet–VAX node. This VMS node runs the Network Control Language (NCL), a general-purpose network management utility. On the DECnet–Mica node, a facility called CMIP (Common Management Information Protocol) processes remote NCL directives. DECnet–Mica includes special local management capabilities to initialize, shut down, and manage the local network node.

Network Event-Logging Server. On a DECnet–Mica node, network events are handled locally by a network event-logging facility. Management utilities on client nodes, such as VMS nodes, can capture these DECnet–Mica events, and log them either to a file on the client node or to a file on the DECnet–Mica node.

Figure 39–1 shows how the components of DECnet–Mica interact. (Note that, though the data-link layer is shown in Figure 39–1, its design is covered in Chapter 19, NI Function Processor. The data-link layer is included here only to complete the figure.)

39.1.3 User Interface

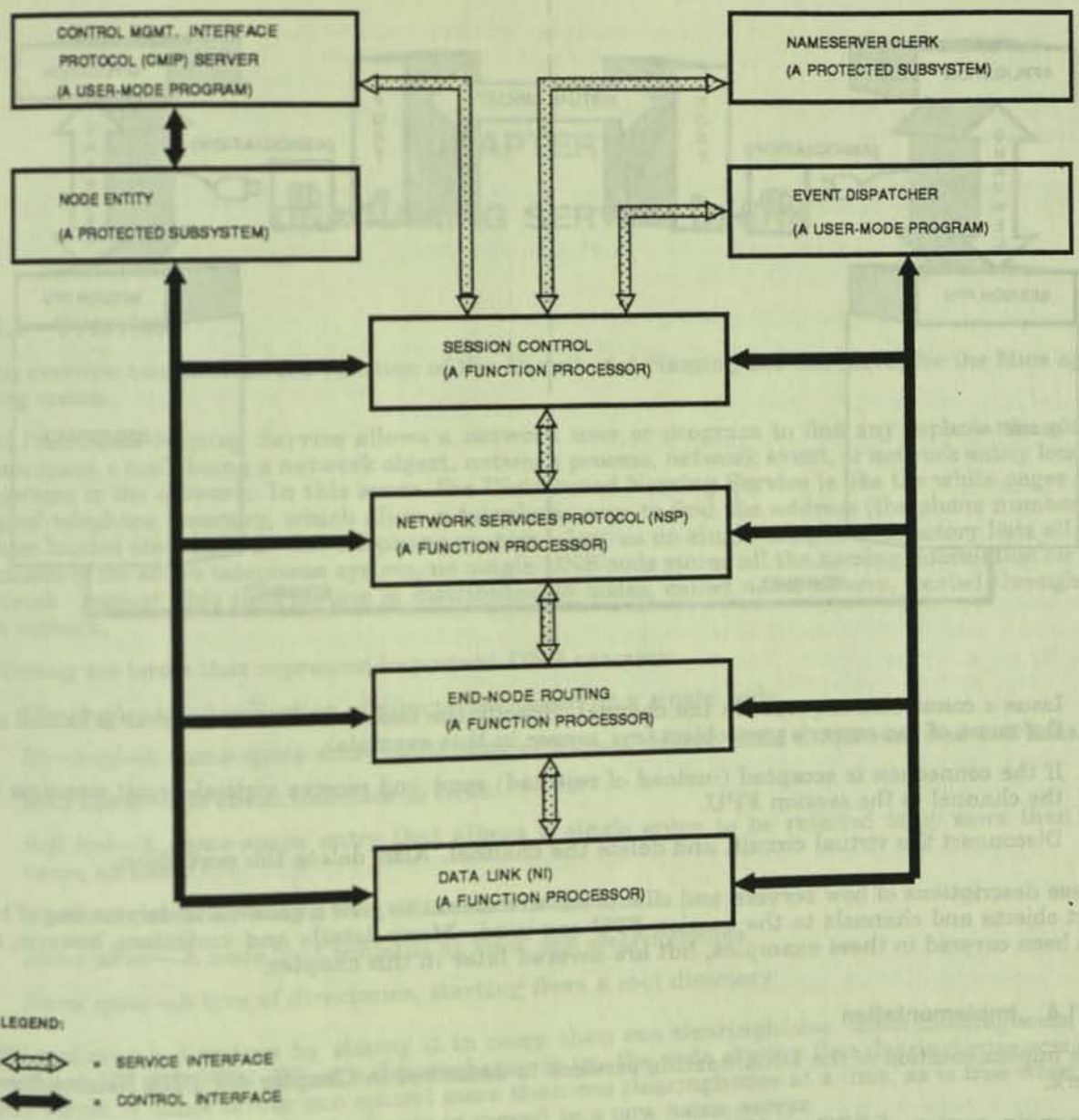
This section summarizes the user interface to DECnet–Mica. The user interface to the DNA naming services is described in Chapter 40, DNA Naming Service Clerk.

The user interface to DECnet–Mica is based on the services of two Mica software components: *port objects* (ports) and I/O channels to the session function processor. Figure 39–2 shows the relationship between port objects and channels. Primarily, a port object represents the data structures within DECnet–Mica used to keep virtual-circuit state. Each port object represents one end of either a virtual circuit or a *potential virtual circuit* (that is, one not yet established). One port object is required for each end of a virtual circuit.

Once a port object is used to establish a virtual circuit, a channel to DECnet–Mica's session FPU is used to send and receive messages via that virtual circuit. Briefly, here are the steps a server thread must go through to create and use a virtual circuit:

1. Create a port, and give it a name (such as *xyz_server*). There are several ways to do this.
2. Wait for a virtual-circuit connection request to be queued to the port. This is done using the executive wait services.
3. Assign a channel to the session FPU, and associate the channel with the port.

Figure 39–1: The Components of DECnet–Mica

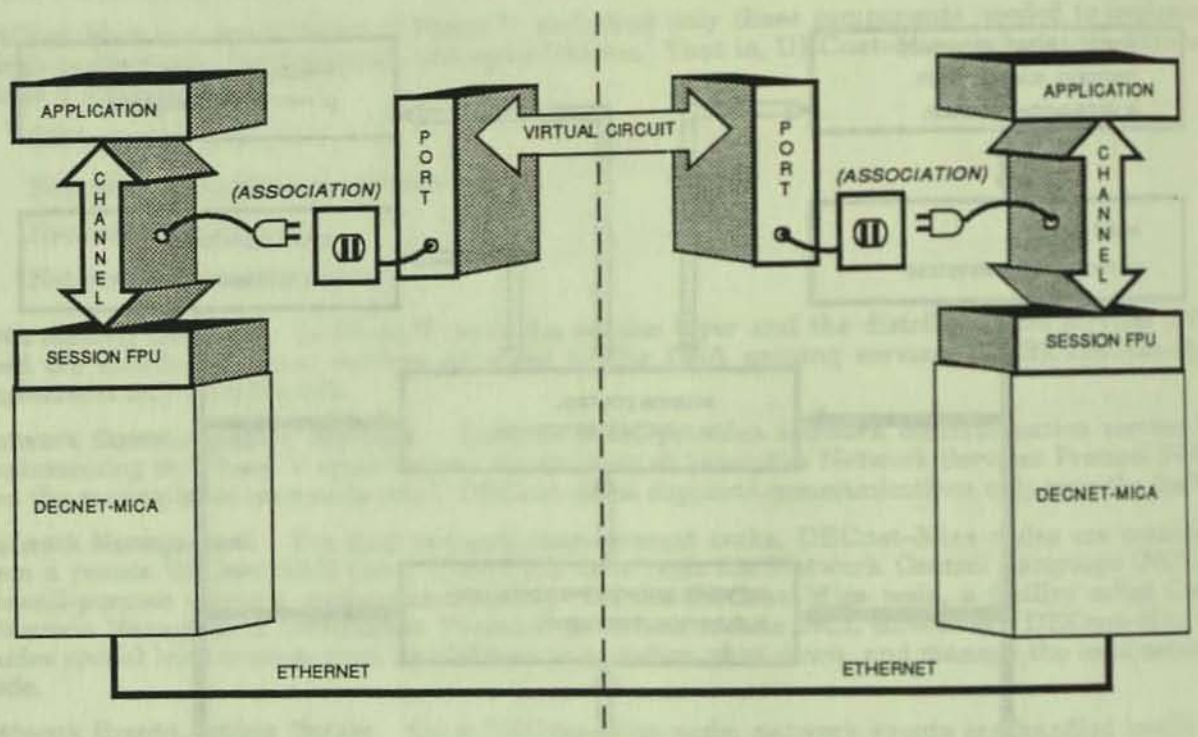


4. Accept the virtual-circuit request via the channel to the session FPU.
5. Send and receive virtual-circuit messages via the channel to the session FPU.
6. Disconnect the virtual circuit, and delete the channel.
7. If the port is to be used for another virtual circuit, proceed at Step 2. Otherwise, delete the port.

To interact with a server thread via a virtual circuit, a corresponding client thread must go through the following steps:

1. Create a port (it does not need a name).
2. Assign a channel to the session FPU, and associate it with the port.

Figure 39–2: Relationship of Ports and Channels in a Virtual Circuit



3. Issue a connection request via the channel. Specify the node on which the server is located and the name of the server's port object (*xyz_server* in this example).
4. If the connection is accepted (instead of rejected) send and receive virtual-circuit messages via the channel to the session FPU.
5. Disconnect the virtual circuit, and delete the channel. Also, delete the port object.

These descriptions of how servers and clients act are meant to give a general understanding of how port objects and channels to the session FPU are used. Many details and variations, however, are not been covered in these examples, but are covered later in this chapter.

39.1.4 Implementation

The implementation of the DNA naming services is described in Chapter 40, DNA Naming Service Clerk.

The main portion of DECnet–Mica (session, NSP, and routing) consists of three independent, but tightly-coupled, function processors. Here, *tightly-coupled* means that the interfaces among these function processors are intended to be as efficient as possible, using special entry points and callbacks as needed.

Both the CMIP server and the event-control components are user-mode programs that use special *exec$request_io* functions to interact with the session, NSP, and routing function processors.

CHAPTER 40

DNA NAMING SERVICE CLERK

40.1 Overview

This overview summarizes the function of the Distributed Naming Service (DNS) for the Mica operating system.

The Distributed Naming Service allows a network user or program to find any tuple in the global name space, a *tuple* being a network object, network process, network event, or network entity located anywhere in the network. In this sense, the Distributed Naming Service is like the white pages of a typical telephone directory, which allow a telephone user to find the address (the phone number) of a user located elsewhere in the telephone system. Just as no single telephone directory lists all the numbers of the entire telephone system, no single DNS node stores all the naming information for the network. Instead, this information is distributed to nodes, called *name servers*, located throughout the network.

Following are terms that represent important DNS concepts:

- *Clearinghouse*—A collection of directories stored on a single node.
- *Directory*—A name-space entry containing entries of objects, child directories, and soft links.
- *DNS Clerk*—The client interface to DNS.
- *Soft link*—A name-space entry that allows a single entry to be referred to by more than one name, an alias.
- *Name*—A character string that refers to an object.
- *Name server*—A node that contains at least one clearinghouse.
- *Name space*—A tree of directories, starting from a root directory.

DNS replicates a directory by storing it in more than one clearinghouse. Each clearinghouse has two states, UP or DOWN. When a clearinghouse is up, the node storing that clearinghouse acts as a name server. A name server can control more than one clearinghouse at a time, as is true when one name server fails and the clearinghouse is moved to a new name server.

All nodes using the Distributed Naming Service have a DNS Clerk. The Clerk contacts name servers that, in some instances, reside on other nodes. The Clerk is the only portion of DNS included in DECnet–Mica for FRS.

For a more detailed description of the DECnet Naming Service, consult the *DNA Naming Service Functional Specification*.

40.1.1 Requirements

The Distributed Naming Service is required for the session layer of Phase V DECnet.

DNS must unambiguously label objects, processes, events, and entities over a distributed system. DNS therefore needs unique identifiers over time and space. To satisfy this need, DNS requires the time system service and the unique-identifier system services. The Distributed Time Server is not implemented; therefore, if the system manager does not set the system time correctly, the time can go backwards.

40.1.2 Functional Interfaces

The Distributed Naming Service has two interfaces—the client interface and the management interface. Both are part of the DNS Clerk.

The client interface allows the following operations:

- Creation of entries in a name space
- Deletion of entries in a name space
- Modification of entries in a name space
- Retrieval of entries in a name space

The kinds of entries affected are network objects, name-space directories, and soft links. The client interface allows the replication of directories in multiple clearinghouses.

The management interface allows control of the name servers, the clearinghouses, and the name space as a whole. All functions of the management interface require privilege, and can be performed only by an authorized user. For management functions affecting only a single name server, this user could be the system manager of the affected node. In contrast, only the network manager should perform functions that affect the structure of the name space. Phase V of the DIGITAL Network Architecture requires that the name server somehow authenticate all management functions performed through the management interface. For FRS we do local authentication on the local node; remote management, however, must have DECnet proxy authentication to the local node before performing privileged functions.

The management interface allows authorized users to perform the following tasks:

- Check the status of a name server
- Turn a clearinghouse on and off
- Create and delete a clearinghouse
- Check the status of a clearinghouse
- Merge and unmerge name spaces
- Turn a name server on and off

40.1.3 Implementation

The DNS Clerk is implemented as a protected subsystem, thereby isolating any problems in the Clerk from the kernel that would be present if the Clerk were implemented as a function processor. The interface to the DNS Clerk is defined as a run-time library (RTL) routine. The RTL routine makes remote procedure calls to the Clerk, which is also on the local node. Performance constraints may later require that the Clerk be implemented as a function processor. By currently defining the interface as an RTL routine, however, we minimize the impact of such a change in design.

CHAPTER 41

DECNET STARTUP, SHUTDOWN, MANAGEMENT, AND LOGGING

41.1 Overview

This chapter covers the following aspects of DECnet–Mica:

- Network management, and event logging
- Network management security
- DECnet–Mica startup
- DECnet–Mica shutdown

41.1.1 Requirements

The aspects of DECnet–Mica covered in this chapter meet the following requirements:

- Allow DECnet–Mica to be managed from a remote node
- Adhere to the network management architecture of the DIGITAL Network Architecture, Phase V [hereinafter, *Phase V*]
- Allow DECnet–Mica to be started locally without network links or terminals, including the console
- Allow DECnet–Mica to be managed from the console

41.1.2 Network Management and Event Logging

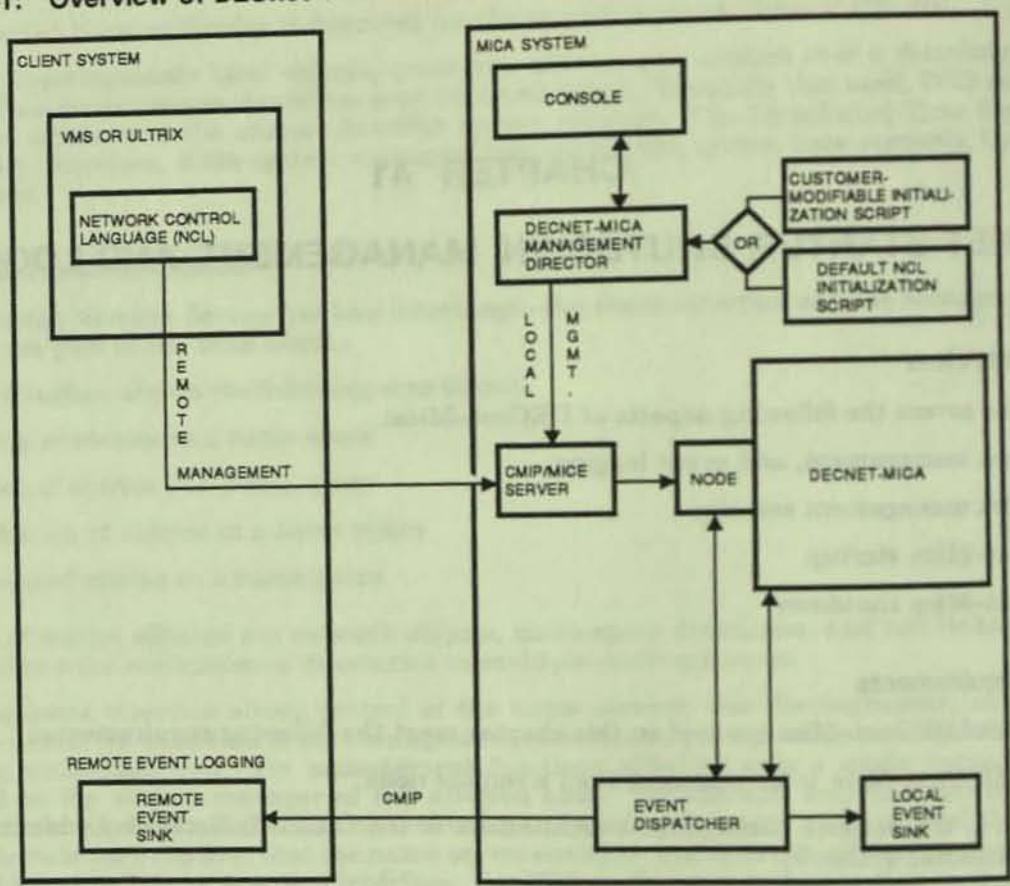
Figure 41–1 shows how the main components of DECnet–Mica network management and event logging relate to one another. Figure 41–2 shows a more detailed view of some of these components. The rest of this section describes components shown in Figure 41–1 and Figure 41–2. For more detailed information on DECnet network management, see the *DNA Network Architecture General Description* and the *DNA Network Management Architecture*.

41.1.2.1 Entities, Directors, and Agents

In Phase V, a network is managed through its entities. An *entity* is a manageable component of a distributed system—a protocol module, for example, is an entity. A discussion of network entities, however, is beyond the scope of this chapter. For details on particular entities, see the network management chapters of the following specifications:

- *DNA Network Control Language (NCL) Specification*
- *DNA Common Management Information Protocol (CMIP) Specification*
- *DNA Maintenance Operations Functional Specification*

Figure 41–1: Overview of DECnet–Mica Network Management and Event Logging



- *DNA Session Control Layer Functional Specification*
- *DNA NSP Functional Specification*
- *DNA Routing Layer Functional Specification*
- *DNA CSMA/CD Data Link Functional Specification*
- *DNA NI Node Product Architecture Specification*

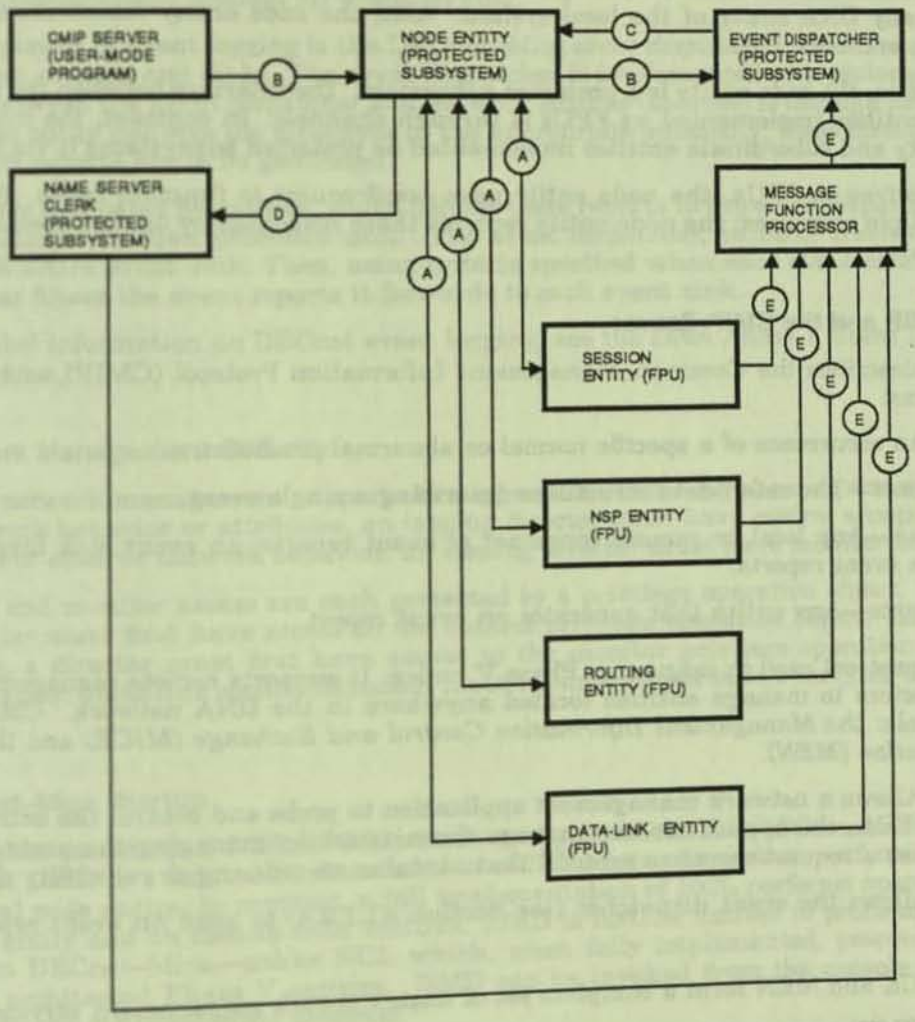
A *director* is a network program that initiates *directives*, which control a network entity. Directives are sent using the Control Management Interface Protocol (CMIP), which is described in a later section. A director is either local (that is, running on the system it manages) or remote. Unlike a *remote director*, a *local director* can perform management operations when the system is not yet part of a network, for example during system initialization.

An example of a director is the *Network Control Language (NCL)*, a command-driven director defined by Phase V. Used to manage DECnet–Mica nodes remotely, NCL provides commands defined by all the manageable entities specified in Phase V. For a complete description of NCL, see the *DNA Network Control Language (NCL) Specification*.

Using the Common Management Information Protocol (CMIP), NCL on one node communicates with the CMIP server on another. The CMIP server, in turn, communicates with the individual entities of its local node.

Each entity contains one or more *agents*. In an entity, the *agent* is the portion that processes management directives received by the entity. That is, the agent is the the management interface between directors and the entity in which the agent resides. The management functions that the agent performs are specified in the DNA specification for the entity.

Figure 41–2: Details of DECnet–Mica Network Management and Event Logging



LEGEND:

- (A) - CHANNEL WITH CMIP AND EVENT DIRECTIVES
- (B) - RPC WITH CMIP DIRECTIVES
- (C) - RPC WITH EVENT DIRECTIVES
- (D) - RPC WITH CMIP AND EVENT DIRECTIVES
- (E) - EVENT REPORTS

41.1.2.2 Node Entity

In Phase V, a *node entity* is a *global entity*, encompassing all other entities of the node. These other entities are therefore *subordinate entities* of the node entity. Roughly equivalent to a local computer system, the node entity is the focus of network management activities within a system; it provides the top-level interface to network management, both local and remote. The node entity forwards directives to any DNA entity of the local system. Also, the node entity maintains the node name, node address, and node UID.

In DECnet–Mica, the node entity is a protected subsystem. The interface between the node entity and subordinate entities implemented as FPUs is through channels. In contrast, the interface between the node entity and subordinate entities implemented as protected subsystems is via intranode RPC.

To issue directives to FPUs, the node entity uses *exec$request_io* function codes. Some directives result in multiple responses; the node entity receives these responses by using a special *exec$request_io* function code.

41.1.2.3 CMIP and the CMIP Server

This section describes the Common Management Information Protocol (CMIP), and introduces the following terms:

- *Event*—An occurrence of a specific normal or abnormal condition.
- *Event report*—The set of data structures describing a single event.
- *Event sink*—Any local or remote consumer of event reports; an event sink displays, stores, or processes event reports.
- *Event source*—Any entity that generates an event report.

CMIP is the protocol used in managing Phase V nodes. It supports remote management operations, allowing directors to manage entities located anywhere in the DNA network. CMIP is composed of two protocols: the *Management Information Control and Exchange (MICE)* and the *Management Event Notification (MEN)*.

- MICE—Allows a network management application to probe and control the entities of a remote system. Thus, the application can manage the network by managing the parts of the network. MICE uses a request/response protocol that contains no ordering or reliability checks.
- MEN—Allows the event dispatcher (see Section 41.1.2.4) to send an event report to an event sink.

Together, MICE and MEN form a complete set of basic network management services.

In Mica, the CMIP server runs as a user-mode program; its interface to the node is through intranode remote procedure calls.

For more detailed information on CMIP, see the *DNA Common Management Information Protocol (CMIP) Specification*.

41.1.2.4 DECnet–Mica Event Dispatcher

Each layer of Phase V—such as routing, NSP, and ISO transport—defines as events certain occurrences, actions, transactions, or conditions. These events are reported, and can be logged to assist in network management. Event logging is the framework for handling these events. The MEN portion of CMIP is the management protocol for event logging.

The major component of event logging is the DECnet–Mica *event dispatcher*, which manages the connections between sources and sinks. The event dispatcher is implemented as a protected subsystem. The interface between the event dispatcher and network entities (sources) is through RPC to the node entity. The node entity delivers the directives to the appropriate subentity; these directives establish events for which reports are to be generated.

Event reports are generated by entities, which deliver these reports to the event dispatcher by writing to a known message function processor unit. The event dispatcher, in turn, creates an outbound stream for each active event sink. Then, using criteria specified when each event sink is set up, the event dispatcher filters the event reports it forwards to each event sink.

For more detailed information on DECnet event logging, see the *DNA Phase V Event Logging Functional Specification.*

41.1.3 Network Management Security

DECnet–Mica network management supports two types of access: *control access* and *monitor access*. To modify network behavior or attributes, an issuing director must have control access. Similarly, to read attributes or observe network behavior, an issuing director must have monitor access.

Control access and monitor access are each protected by a privilege operation object. To get control access, a director must first have access to the control privilege operation object. Likewise, to get monitor access, a director must first have access to the monitor privilege operation object. Each DECnet–Mica agent (data-link agents included) must validate access before carrying out each CMIP directive.

41.1.4 DECnet–Mica Startup

The software that starts up a DECnet–Mica node entity is called the *DECnet–Mica management director (DMD)*. DMD is a limited implementation of NCL, being limited to performing operations only on the local node entity. In contrast, a full implementation of NCL performs operations both on the local node entity and on remote node entities. DMD is further limited to processing commands relating only to DECnet–Mica—unlike NCL which, when fully implemented, processes commands relating to all architected Phase V entities. DMD can be invoked from the console by entering a command within the console control program.

DMD runs at boot time, and configures the network as specified in a script file, of which the system has two versions: *(1)* a default read-only version, and *(2)* a customer-modifiable copy of the default version. A customer can copy the modifiable file to a client node, edit the file, then copy it back to the DECnet–Mica system. Then, through system management, the network can be started from either the default script or the customer-modified copy

To help ensure that customers can start DECnet–Mica, even if the modifiable startup script is corrupted, the following precautions are taken:

- If an error is detected while the network is starting from a customer-modifiable script, the partially started network is shut down. DECnet–Mica is then restarted, using the default initialization script. After the network is properly started, DECnet–Mica notifies other processes in the system that DECnet is up and ready for use.
- When copied back to the Mica system, the customer-modifiable script is compiled and checked for errors of syntax or range. This checking is performed either by a consistency checker being developed by the DECnet–VAX group or by a similar tool developed by the DECnet–Mica group.

41.1.5 DECnet–Mica Shutdown

The DECnet–Mica management director (DMD) not only starts a DECnet–Mica node, but also shuts it down. The DMD performs an orderly shutdown of all the entities of the network. Note that some network processes, such as the DECnet–Mica node entity and the DNA naming services clerk, must retain information despite shutdowns. These processes keep this information on the system disk, where it can be accessed at the next system startup.

41.1.6 Issues

1. The *DNA Phase V Event Logging Functional Specification* has not gone to field test, so its design is subject to change.
2. The process of creating a node at system installation is not understood. We do not understand how a system starts off with a name and address, then publishes them in the name server.

CHAPTER 42

QUARTZ INTERPROCESS COMMUNICATION

42.1 Overview

Mica provides an interprocess communication (IPC) system that specifically addresses the interprocess communication needs of Quartz. Message queues are the primary mechanism for interprocess communication on Quartz systems. In the message queue model, processes that want to send messages queue them on a *source queue*. Messages may then be retreived from an associated *sink queue*. The following sections discuss the message queue model in greater detail.

42.1.1 Requirements/Goals

Quartz has the following requirements for an IPC mechanism:

- Provide an interface transparent mechanism for communicating with local and remote processes.
- Support messages of any size (up to the limit imposed by the size of the memory region created for buffering messages).
- Allow more than one source queue to be associated with a sink queue.
- Allow a sink queue to have more than one reader.
- Present messages at the sink queue in the same order as messages at each source queue. If a sink queue has more than one source queue, then message order need not be preserved among the source queues.
- Provide a flow control mechanism to control the message rate of delivery at the sink queue.

42.1.2 Non-Goals

It is a non-goal to protect the contents of a message while residing in the shared memory region. Since the region is shared between processes, the complexity involved to provide data integrity would cause a performance degradation of the IPC mechanism.

42.1.3 Functional Description

To establish communication among processes on a Quartz system, a *message region object* is created, which is a portion of memory shared among source and sink queues. It provides memory for storing messages.

Sink queues are created first, followed by the creation of their associated source queues. The creation of a source queue establishes a link between the source and sink queue. This action creates a *web*, an interconnected set of queues. If the creation of a subsequent source queue specifies an existing sink queue, the new source queue joins the existing web. Note that this implies a many to one relationship between source and sink queues.

Before interprocess communication can occur, a message gate object (hereafter referred to as a *gate*) must be created for each process using the web. Gate creation can occur any time after region creation. The Message Gate Creation service maps the specified memory region into the address space of the process issuing the call. A gate must be explicitly bound to a source or sink queue to join a web, and to subsequently send or receive messages.

After the creation of the memory region, the creation of the necessary queues, the creation of the corresponding gates, and the binding of gates to source and sink queues, messages can be sent and received. Messages can be transferred in several different ways. Messages are stored in buffers in the memory region associated with the gate. The buffers may be preallocated. A thread can later reference a buffer as part of the process address space. A message may also be copied into a buffer that is part of the region. In addition, a thread may request to use a message queue in a protected mode, forcing all messages to be copied, i.e., no user mode access to the region. Note that this mode is intended for debugging purposes and will incur severe performance degradation.

Gates support an end-of-stream operation. When all gates associated with a source queue have set end-of-stream, the source queue's state is set to end-of-stream. When all source queues in the web have entered end-of-stream, all members of the web transition to web end-of-stream. Any requests to send a message while a source queue is in the source queue end-of-stream state result in the message not being sent. In this case, end-of-stream status is returned to the caller. Any requests to receive a message while a sink queue is in the web end-of-stream state result in end-of-stream status being returned to the caller.

42.1.4 Design

Quartz message queues consist of three objects: regions, queues, and gates.

42.1.4.1 Message Region Object

A *message region object* is a portion of memory shared among source and sink queues; it provides memory for storing messages. The minimum allocatable message size is a run-time constant with a default value of 128 bytes.

42.1.4.1.1 Functional Interface

The Message Region Creation object service is used to create the region. The object architecture requires that kernel mode objects not be placed in user mode containers. When the Message Region Creation service is called, the previous mode of the calling thread is temporarily changed to user, in order to create a user-mode memory section. Message regions are deleted using *exec$delete_object*.

Since the pages in the section are user read/write, all message allocation data structures reside in nonpaged pool. Allocation and initialization of these data structures occurs when the region is created.

Executive service routines are used to allocate and deallocate message space from the region. These services are used by the gate system services as a result of a request by a thread to allocate (or deallocate) a message associated with a gate.

42.1.4.2 Message Queue Object

This object provides one end of a uni-directional virtual circuit between itself and another queue object. It provides flow control, and maintains a list of messages to be sent or delivered, depending on the type of message queue.

Message queues may be waited on using the Mica wait services. When a thread waits on a source queue, a change in the queue's state causes the thread to unwait. Table 42–1 describes the various message queue states. Table 42–2 and Table 42–3 describe the transitions between source and sink queue states. Sink queues may be waited on to detect the arrival of a new message. A kernel event object is used as the dispatch object.

Flow control between source and sink queues is accomplished by the use of a message-based credit window mechanism associated with each source queue. No messages are queued for delivery at the sink queue, when the credit value is zero. When a message is removed from the sink queue, the credit value of the source queue that sent the message is incremented.

The delivery characteristics of a message queue may be modified by two thresholds:

1. Restart Sender—When a thread attempts to send a message on a source queue with no message credits, the thread is placed in a resource wait state. It will remain in this state until a specified number of message credits are again available.
2. Restart Receiver—When no messages are available from a sink queue, the sink queue will remain unsignaled until there are a specified number of messages queued for delivery in the sink queue.

Table 42–1: Message Queue States

State	Valid For	Description
CONFIGURE	Source and Sink Queues	This is the initial state for all queues. The thresholds and credit limit associated with a queue can only be changed while the queue is in this state. A source queue's signal state is cleared when the queue enters this state. A sink queue's signal state is signaled when in this state. Waiting on a sink queue while in this state is meaningless, that is, the sink queue's signal state does not change implicitly while in the CONFIGURE state.
SOURCE QUEUE READY	Source Queues	This state provides an intermediate point for web startup synchronization. The source queue's signal state remains cleared.
WEB RUNNING	Source and Sink Queues	Message transmission may commence upon entering this state. A source queue's signal state is signaled. A sink queue's signal state changes whenever messages are present to be read, or when the last message is read out of the sink queue.
SOURCE QUEUE END OF STREAM	Source Queues	This state provides an intermediate point for web shutdown synchronization. The source queue's signal state is cleared.
WEB END OF STREAM	Source and Sink Queues	This state signifies that messages will no longer be sent by source queues. The source queue's signal state is cleared. The sink queue's signal state is signaled.
FAILURE	Source and Sink Queues	This state is entered if the executive service routines detect a catastrophic failure. Transitions out of this state are not allowed.

Table 42–2: Source Queue State Transitions

From	To	Description
CONFIGURE	SOURCE QUEUE READY	A source queue enters this state after it is configured. The *exec$set_message_queue_state* service is used to make this transition.
SOURCE QUEUE READY	CONFIGURE	A source queue enters this state when it needs to be reconfigured. The *exec$set_message_queue_state* service is used to make this transition.
SOURCE QUEUE READY	WEB RUNNING	Source queues make this transition when the web's sink queue has transitioned from CONFIGURE to WEB RUNNING. Source queues can now perform message transmission.
WEB RUNNING	SOURCE QUEUE END OF STREAM	A source queue reaches this state when all gates associated with the source queue have set end-of-stream. A gate is set to end-of-stream by using the *exec$set_end_of_stream* system service.
SOURCE QUEUE END OF STREAM	WEB END OF STREAM	When all source queues have reached the SOURCE QUEUE END OF STREAM state, then all queues in the web move to this state.
Any State	FAILURE	Any catastrophic failures detected by the executive service routines will cause a transition into this state. Transitions out of this state are not allowed.

Table 42–3: Sink Queue State Transitions

From	To	Description
CONFIGURE	WEB RUNNING	A sink queue can make this transition only when all its source queues are in the SOURCE QUEUE READY state. This transition places the source queues in the WEB RUNNING state and is performed by issueing the *exec$set_message_queue_state* service.
WEB RUNNING	WEB END OF STREAM	As the last source queue in the web transitions from WEB RUNNING to SOURCE QUEUE END OF STREAM state, all queues in the web move to WEB END OF STREAM.
Any State	FAILURE	Any catastrophic failures detected by the executive service routines will cause a transition into this state. Transitions out of this state are not allowed.

Figure 42–1 and Figure 42–2 depict the finite state machines for source and sink queues.

Figure 42–1: Source Queue States

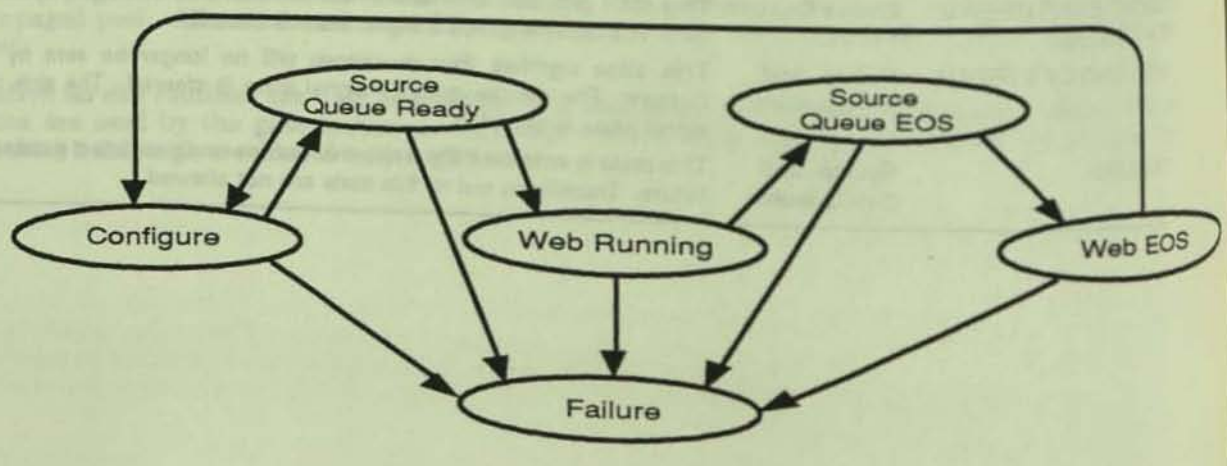
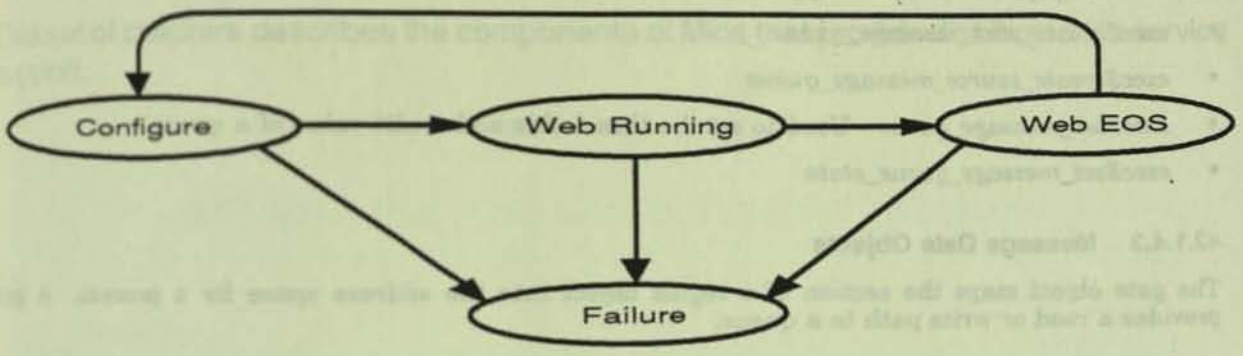


Figure 42–2: Sink Queue States



42.1.4.2.1 Remote Queues

Source and sink queues on Mica systems use Systems Communications Services (SCS) to communicate (see Chapter 17, System Communication Services). Included in the code that implements queue objects is a section that implements a SCS system application (SYSAP). In this section, the term *cluster* refers to Mica systems that are members of a SCA cluster, and the term *node* refers to a Mica system.

The following list details the major pieces of the SYSAP to implement remote message queues:

- As the message queue object type descriptor is created, a channel to the SCS function processor is obtained. The queue SYSAP registers with the SCS directory services, and enables the handling of incoming connect requests from other queue SYSAPs.

  A channel is opened to the notification message FPU (see Chapter 34, Configuration Management Software) requesting messages that indicate new nodes entering the cluster. The SCS function processor is the source of these messages. This information prevents two nodes entering the cluster from missing each other's entry in their configuration services listing.

- SCS can provide multiple paths between nodes in the cluster. As SCS nodes are encountered, each queue SYSAP builds a fully connected graph to all nodes running the queue SYSAP. When a path fails, the queue SYSAP is notified and removes that path from the list of paths to the failed node. If this results in the last path being removed, then all webs between the two nodes move into the FAILURE state.

- Source queues wanting to associate with a remote sink queue must designate the remote node and sink queue name. A sink queue indicates its ability to accept remote associations via a parameter in the sink queue creation service. The name of the sink queue is cached by the queue SYSAP. Association between the two queues occurs if the sink queue name is in the remote queue SYSAP's list of sink queues.

- Changes in the state of the web are propagated to queues on other nodes via SCS sequenced messages. A change in source queue state sends a message to its corresponding sink queue. Changes in sink queue state may result in more than one message being sent to update the state of all associated remote source queues.

- Messages are transmited by first mapping the send and receive buffers with SCS. The sink queue SYSAP issues a SCS block read function.

42.1.4.2.2 Functional Interface

The following system services support Quartz interprocess communication:

- *exec$create_sink_message_queue*
- *exec$create_source_message_queue*
- *exec$set_message_queue*—Used to set the thresholds and credit value of a queue.
- *exec$set_message_queue_state*

42.1.4.3 Message Gate Objects

The gate object maps the section of a region object into the address space for a process. A gate provides a read or write path to a queue.

42.1.4.3.1 Functional Interface

The following system services support Quartz interprocess communication:

- *exec$create_message_gate*
- *exec$allocate_message_buffer*—Returns a pointer to a message buffer in the region.
- *exec$deallocate_message_buffer*—Returns a message buffer to the region. The calling thread passes a pointer to the message.
- *exec$bind_gate_to_queue*—This service associates a gate with a message queue, and consequently determines the types of operations that may be performed on the gate, e.g., source gates can only send, sink gates can only receive.
- *exec$unbind_gate_from_queue*
- *exec$send_allocated_message*
- *exec$receive_allocated_message*—If no message is available, then status is returned to the user.
- *exec$send_user_buffer*—Requests that a message not in the region be sent. A message buffer from the region is allocated to hold a copy of the user buffer. The message is transmitted in the normal fashion.
- *exec$receive_user_buffer*—If a message is available for delivery at the sink queue, then it is copied into the user specified address and deallocated.
- *exec$set_end_of_stream*

Distributed File Services

This set of chapters describes the components of Mica that provide distributed file service support.

CHAPTER 43

DISTRIBUTED FILE SERVICE INTRODUCTION

43.1 Overview

The Distributed File System (DFS) architecture Version 1.0 is a standard set of protocols for file access between Digital systems. The VMS implementation (VMS DFS) is provided by a layered product which is already available to customers. This document describes the implementation of DFS for the Mica operating system (Mica DFS). An implementation of the DFS Version 1.0 architecture for ULTRIX is not planned to be available at the time of the first release of Mica DFS.

Most user programs, applications and file utilities do not have to be changed to access remote files using DFS. There are some restrictions; these are discussed in Section 43.1.4. For each supported operating system, DFS provides a component that implements the same block-level I/O system interface as the local file system. Within the I/O system, dispatching based on the device name directs file I/O requests to either the local file system or DFS.

Mica DFS operates with both Glacier and Cheyenne, but it is supported for customer use with Glacier only.

43.1.1 Goals

The goals of the first release of Mica DFS are:

- To allow VMS systems to transparently access Mica files, and to allow Mica systems to transparently access VMS files, subject to the restrictions in Section 43.1.4.
- To provide a basic distributed file system, from which a network transparent file environment can be provided on Glacier.

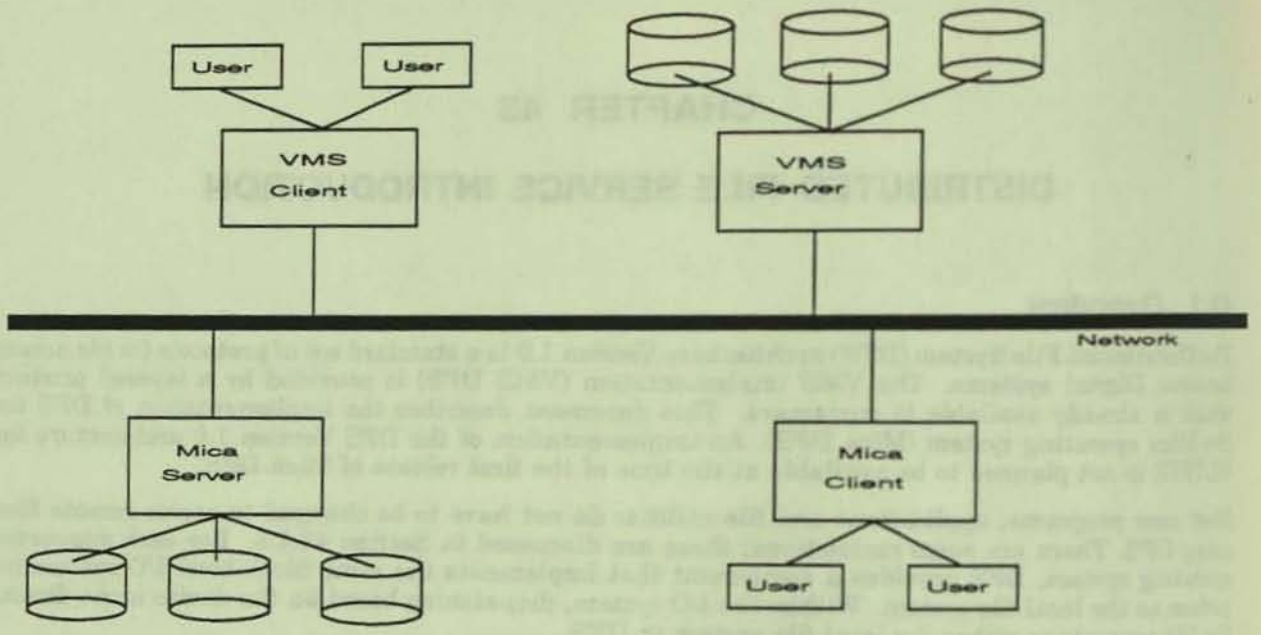
43.1.2 Model

DFS is organized using a client/server model. The *client* is the software that acts on behalf of the user, accepting file requests, formatting them as appropriate, and sending them to the server. The *server* is the software that receives file requests from clients and executes them. The server process for a particular file is on the same system as that file. Note that the VMS DFS product can act as both a client and a server, and that Mica DFS can also act as a client and a server.

Figure 43–1 shows a network of Mica DFS and VMS DFS clients and servers.

The DFS Version 1.0 architecture provides a *virtual block oriented* file service. The protocol being exchanged between systems describes file system functions that manipulate disk blocks within particular files. The DFS Version 1.0 file protocol is an RPC implementation of the VMS file system XQP QIO interface.

Figure 43–1: DFS Clients and Servers in a Network



The DFS protocol does not express logical-block (volume-relative) requests or record-oriented requests. The DFS protocol is completely unrelated to the Data Access Protocol (DAP) used by VMS today. DAP is a difficult-to-decode, record-oriented protocol for use between heterogeneous systems. DFS is a specialized, block-oriented protocol which is optimized for VMS-to-VMS file communication.

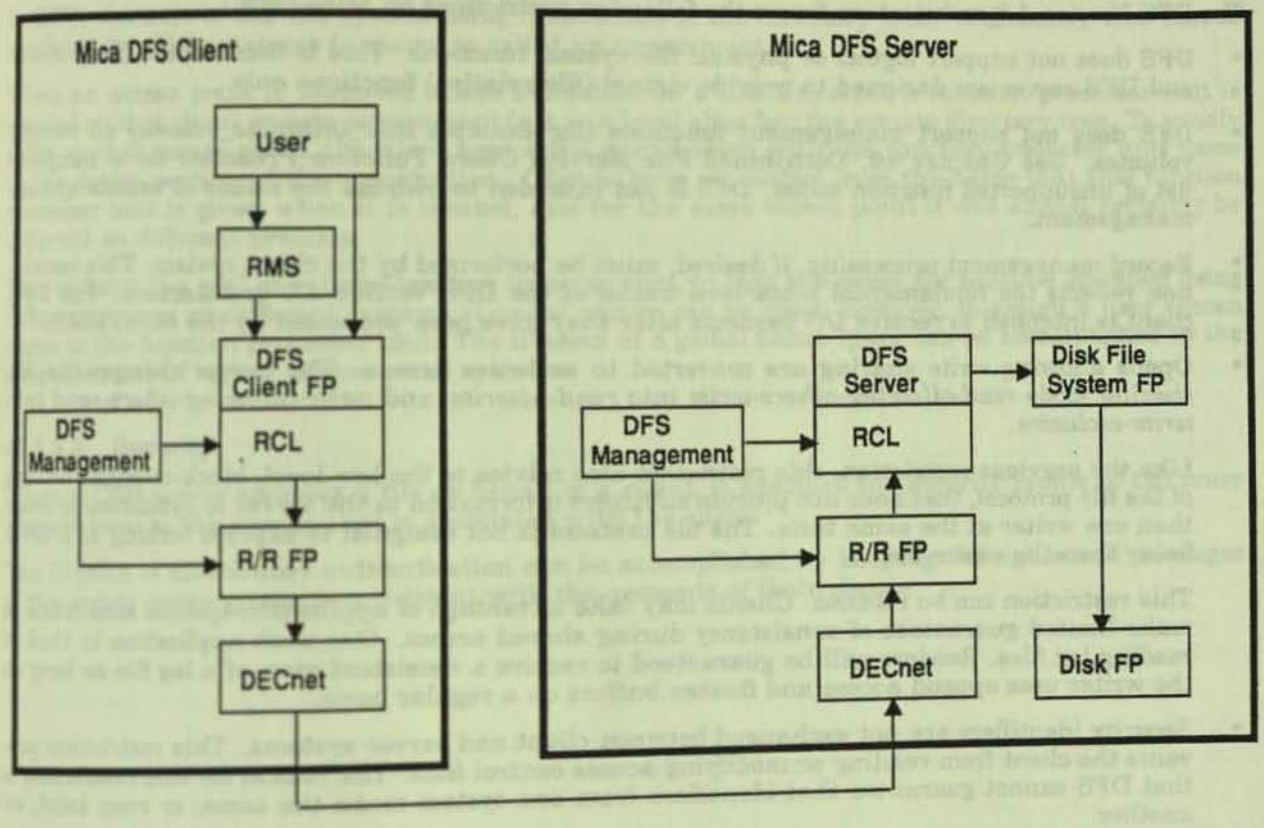
43.1.3 Components

Figure 43–2 is an overview of the Mica DFS system. The five major components of the system are

- the DFS client function processor
- the request/response command language
- the request/response function processor
- the DFS server
- the DFS management component

The Mica DFS *client function processor* (client FP) belongs to the disk file function processor (DFFP) class (see Chapter 24, Disk File System Function Processors), and thus presents the same interface as the disk file system function processor (see Chapter 25, Files–11 ODS2 Function Processor). Units of the client FP represent volumes on remote systems instead of local devices. Higher levels of software, for example, RMS and BACKUP, are not generally aware of whether they are using a local device unit or a DFS client unit. When operations are issued to a DFS client unit, the client FP generates a message describing the function and sends it to the server.

Figure 43–2: Mica DFS System



The *request/response command language* (RCL) is a subroutine library that provides argument marshaling and remote procedure call (RPC) services. The DFS client FP uses this layer to encode requests and to decode responses; the DFS server uses RCL to decode requests and to encode responses. RCL, in conjunction with a reliable communication transport, provides an RPC run-time system for use by DFS clients and servers.

The *request/response function processor* (R/R FP) provides a request/response-oriented communication service that is used by RCL. The Mica implementation of the R/R FP is designed for compatibility with the corresponding VMS DFS R/R component. In general, a request/ response-oriented communication service is desirable because it is the natural transport for RPC-style protocols such as the DFS file protocol, and because it permits a high-performance implementation on a local-area network. The first release of the Mica R/R FP uses DECnet-Mica exclusively as its network transport. See Chapter 39, Network Services, for more information on DECnet-Mica.

The Mica DFS *server* receives and decodes client messages using RCL, and then performs the actual requests on the volumes represented by DFS client FP units. The server is implemented as a user-mode, multithreaded program. Server functions are invoked as remote procedure calls that manipulate files being managed by the server.

Mica DFS *management* allows authorized individuals to control the operation of the DFS software. Mica DFS management is implemented as part of Mica system management. Common management operations include creating a client FP unit to represent a remote volume, establishing R/R communication characteristics, and allowing a local volume to be served by a DFS server.

43.1.4 Planned Restrictions

The DFS Version 1.0 architecture forces the following restrictions on Mica DFS:

- DFS does not support logical or physical file system functions. This is because the file protocol and DFS server are designed to provide virtual (file-relative) functions only.
- DFS does not support management functions (for example *io$c_initialize_volume*) on remote volumes. See Chapter 46, Distributed File Service Client Function Processor for a complete list of unsupported function codes. DFS is not intended to address the issues of remote system management.
- Record management processing, if desired, must be performed by the client system. This restriction reflects the fundamental block-level model of the DFS Version 1.0 architecture. The DFS client is intended to receive I/O requests after they have been processed by the client RMS.
- Opens allowing write sharing are converted to exclusive access. The server changes the file sharing mode *read-allowing-others-write* into *read-nowrite*, and *write-allowing-others-read* into *write-exclusive*.

  Like the previous restriction, this restriction also relates to the low-level, block-oriented nature of the file protocol, that does not provide sufficient information to the server to synchronize more than one writer at the same time. The file protocol is not designed to express locking at a level lower than the entire file.

  This restriction can be relaxed. Clients may take advantage of application-specific semantics to make limited guarantees of consistency during shared access. One such application is that of reading log files. Readers will be guaranteed to receive a consistent view of a log file as long as the writer uses append access and flushes buffers on a regular basis.
- Security identifiers are not exchanged between client and server systems. This restriction prevents the client from reading or modifying access control lists. The reason for this restriction is that DFS cannot guarantee that identifiers from one system mean the same, or even exist, on another.

43.1.5 Network Transparency

Transparent execution of programs on Glacier requires that the server provide the same execution environment as the client. Implementations of the DFS version 1.0 architecture do not provide global file naming or distributed security. Through consistent management between systems, the following aspects of transparency can be provided:

- *name transparency* - File volume names have the same meaning on all systems
- *volume-location transparency* - The location of a file volume can be changed without affecting its name
- *application-location transparency* - Applications can perform the same file operations independent of the system on which they are executed

43.1.5.1 Naming

In DFS, the unit of the file system being distributed is the directory tree. A directory tree that is available for other systems to mount is called an *access point*.

When an access point is mounted (made available) on a client system, a function processor unit is created on that client system to represent (act as a local alias for) the remote directory tree. To specify a file on that access point, the client uses a file specification with the function processor unit name in the device portion of the specification. Clients have no control over the name that this function processor unit is given when it is created, and for the same access point it will almost certainly be different on different systems.

This violates the goal of volume-location transparency, in that the same file must be specified using different names on different systems. Logical names can be used, however, to assign a user chosen name to the function processor unit. The illusion of a global name space can be accomplished by the consistent use of logical names on all systems.

43.1.5.2 Security

When a DFS server accesses a file on behalf of a client, it assumes the security profile of the proxy account local to the server, that is associated with the client.

The illusion of distributed authentication can be accomplished by keeping the rights and privileges of the server proxy accounts consistent with the accounts of their clients.

CHAPTER 44

DISTRIBUTED FILE SERVICE MANAGEMENT

44.1 Overview

This chapter discusses management of the Mica distributed file system (Mica DFS) software.

Mica DFS is composed of three major components: the client function processor, the server, and the request/response function processor. See Chapter 43, Distributed File Service Introduction for more information. Management of Mica DFS consists of starting these components, and controlling them while they are running.

Mica DFS management has the following goals:

- To minimize the number of tuning parameters. This is accomplished by having the software adjust itself according to its environment. The number of threads used by a component, for example, is controlled automatically.
- To minimize the need to tune other components. For example, although the request/response (R/R) function processor uses DECnet, it is a goal to minimize or eliminate the need to modify DECnet performance parameters as part of R/R management.

There are no architected management entities and attributes for DFS like there are for DECnet Phase V.

44.1.1 Restricting Access to Management Operations

Management operations are performed on the two Mica DFS function processors using I/O system requests. Access control lists on I/O system objects prevent unauthorized users from performing management operations. See Chapter 8, I/O Architecture, for more information.

Mica system management and the Mica DFS server are implemented as separate user-mode processes on the same system. The request/response function processor (R/R FP), which is already used for DFS network communication, will also be used in a local mode to accomplish inter-process communication.

Mica system management directs the Mica DFS server to perform management operations using an R/R RPC message protocol. Since R/R does not provide authentication of senders, this server management protocol is responsible for providing the sender's identity to the server. Mica security is then used to authenticate the sender and restrict his operations as necessary. See Chapter 10, Security and Privileges, for more information.

44.1.2 Startup

The request/response and DFS client function processors are loaded automatically with other function processors during system boot. After being loaded, each function processor is called at its initialization entry point. Function processors do not need to be started explicitly. The order in which the request /response and client function processors are started does not matter.

Mica DFS is started relatively late in the sequence of startup operations. See Chapter 34, Configuration Management Software, for a discussion of initial device configuration. The Mica *startup program* is responsible for starting the Mica DFS server, and for calling a Mica DFS specific startup routine to establish the DFS client and server environments.

The Mica DFS startup routine reads the Mica DFS *configuration file* to determine the previous environment to restore. The startup routine then issues management operations to the various Mica DFS components according to the contents of that file. The configuration file is updated whenever a management command is issued interactively that changes the state of Mica DFS.

44.1.3 Monitoring

Each Mica DFS component collects performance data and statistics that can be displayed using the Mica monitor utility. System management SHOW commands are not used to display performance data.

The two Mica DFS function processors adhere to the standard data collection interface as defined in Chapter 36, Performance Monitor. The Mica DFS server, which is implemented as a user-mode server, provides a special message interface to retrieve performance data.

44.1.4 DECnet Name Service

Mica DFS uses the DECnet name service (DNS) to register and look up access point names. DNS associates an *access point name* with a particular remote volume and the location of its DFS server. DNS provides a global namespace for the consistent naming of access points regardless of the location of clients and servers.

Management of DNS is discussed in Chapter 40, DNA Naming Service Clerk.

44.1.5 Management of Files Accessed Through DFS

File management operations, for example directory creation and file copying, work transparently for files on DFS devices as well as local devices. Volume management operations, for example volume initialization and disk quota maintenance, do not work transparently on DFS devices. See Chapter 32, System Management, for a discussion of remote volume management.

44.1.6 System Management

Interactive control of Mica DFS is provided by Mica system management. System management in general is discussed in Chapter 32, System Management.

44.1.6.1 Client Function Processor

Mica system management provides the following commands to control the client function processor:

Command	Description
MOUNT/REMOTE	Invoke the mount utility to create a client unit
DISMOUNT/REMOTE	Invoke the mount utility to remove a client unit
SET CLIENT	Set DFS specific client device unit characteristics
SHOW CLIENT	Show DFS-specific client device unit characteristics

44.1.6.2 Server

Mica system management provides the following commands to control the server:

Command	Description
START SERVER	Start the Mica DFS server
STOP SERVER	Stop the Mica DFS server
ADD ACCESS_POINT	Make a volume available for remote access through the Mica DFS server; the access point is then registered with DNS
REMOVE ACCESS_POINT	Make a served volume unavailable; the access point name is then removed from DNS. Existing mounts may or may not affected, depending on the qualifiers present
SHOW ACCESS_POINT/LOCAL	Show access points that are currently being served
SET SERVER	Modify server characteristics
SHOW SERVER	Show server characteristics
SHOW ACCESS_POINT/REMOTE	Show the access points that are known to the name service and match the provided wildcard specification

The Mica DFS server depends on Mica security to authenticate remote clients and to provide impersonation services. See Chapter 10, Security and Privileges, for information on how the authentication database is managed.

44.1.6.3 Request/Response Function Processor

Mica system management provides the following commands to control the request/response function processor:

Command	Description
SET COMMUNICATIONS	Set communication characteristics
SHOW COMMUNICATIONS	Show communication characteristics

The Request/Response Function Processor (R/R FP) uses a single Function Processor Unit (FPU) for all readers and writers. This FPU is created automatically when the R/R FP is initialized.

CHAPTER 45

DISTRIBUTED FILE SERVICE COMMUNICATION FUNCTION PROCESSOR

45.1 Overview

The Request/Response Transaction Protocol (RR) and the Request-Response Command Language (RCL) together compose a simple RPC communication interface for the Mica Distributed File System (Mica/DFS). RR is a Mica function processor. It accepts communications requests from the DFS client function processor, DFS server, and possibly other function processors and users, then makes requests to the function processor that represents the session layer of DECnet. RCL is a collection of subroutines, called by higher-level function processors and threads, used for formating and packing messages, sending them to a remote node using a call to RR, and unpacking incoming messages from remote nodes.

Versions of both RR and RCL currently exist for VMS DFS. The Mica RR development effort is largely one of implementing an already architected communication interface. Although deficiencies have been recognized in the current RR/RCL design, the Mica/RR and Mica/RCL implementations is constrained to be compatible with the current VMS design. Design changes will be considered for a second, post-FRS version of RR/RCL.

45.1.1 RR

RR is a simple transaction-oriented, intermediate-level communications protocol. The transaction model is that of simple request/response paired communications between clients and servers. A client may make repeated requests within the framework of a common context. Each request/response pair is called a transaction and is identified with a unique transaction ID; a series of transactions within a shared context is referred to as a session and identified with a unique session ID. RR is an intermediate protocol in that it is layered on top of lower-level transport protocols, and its users establish an additional protocol layer on top of RR. This top-level protocol might consist simply of a defined set of functions and calling standards, or a more elaborate set of interactions.

Any number of clients on a particular node (up to a reasonable, parameterized limit) can establish sessions with a server on a remote node. The establishment of sessions, and the transaction exchanges within them, all are multiplexed over a single connection between the two nodes. The connection represents a reliable communications path between nodes, and currently is obtained using a DECnet session-level logical link. The RR architecture is largely independent of how reliable communications are implemented, but does require the properties of logical links for reliable message transfer, congestion control, data segmentation, routing, error recovery at the message level, and node authentication.

45.1.1.1 Interface to Higher-Level Function Processors and Threads

RR can be viewed, from the standpoint of higher-level entities using it, as a very simple set of functions for effecting transactions:

- Open_Port

  This function is used by either a client or server to create an RR port for communications with a remote server or client, respectively. The port is simply an access path for the client or server to RR, and is represented in the Mica/RR implementation by a I/O channel. If the port is being created for use by a server, then the server's availability is made known to remote clients (via the DECnet name server).

- Close_Port

  Either a client or server port is closed using this function, preventing further access to the port. In both cases, appropriate actions are taken to gracefully terminate communications activities.

- Access_Connection

  A client uses this function to establish a connection to a remote server. An actual session-level logical link is created only if one had not been previously established. Once established, logical links are maintained on a semipermanent basis. The basic asymmetry in the protocol is reflected in this function's availability only to clients. Servers do not request establishment of connections or sessions.

- Session_Transaction

  Clients use this function to make all requests. in one call, a session can be opened, a transaction requested, and the session closed. This translates into a single RR message, saving message and packet overhead for simple transactions. In more complex cases, this function can be used to establish a session, subsequently closed by a later call, with the unique session ID supplied as an argument. Sessions allow the client-side user to identify a transaction as belonging to a group of transactions, which the server may then choose to process within a common context. Requests are received by a server in the order in which they are made by a remote client.

- Request_Receive

  A server uses this function to receive requests from remote clients. Requests are identified by transaction, session, and connection IDs. As noted in Section 45.1.1.4, there are design alternatives for processing incoming requests for servers.

- Response_Transmit

  The server uses this function to respond to requests from remote clients.

Detailed description of these functions, their arguments, and corresponding data structures and associations are included in the design portion of this chapter.

45.1.1.2 Interface to DECnet Session Layer

In the current VMS implementation, RR interfaces to the DECnet session layer through a set of thirteen functions. A logically equivalent, although probably simpler interface will be designed for Mica/RR. This interface will be strongly influenced by the design considerations and issues described in Section 45.1.1.4. The interface must allow RR to perform the following functions:

- Open and maintain connections
- Accept and reject connection requests
- Obtain status information
- Transmit and receive data over the connections

- Allocate and return message buffers

45.1.1.3 Interface to System Management

RR provides a system management interface that allows read access to RR's state information and causing those state changes in RR necessary for effective management. For example, the management interface will allow reading and changing the RR parameters for setting maximum limits on connectivity and the number of outstanding transactions. More complex operations, including manipulating internal data structures, will be provided as needed for debugging and management purposes. The management interface will be discussed in detail in Chapter 44, Distributed File Service Management.

45.1.1.4 Implementation

RR is implemented as a multithreaded Mica function processor. On intialization, RR creates a RR function processor unit, to which separate channels may be connected by the DFS client FP, the DFS server, and by any other function processor or system thread that uses RR.

45.1.1.4.1 Data Structures

In the current VMS implementation, RR manages an internal database containing separate data structures describing each transaction, session, and logical link, as well as each client and server port. Mica/RR manages equivalent data structures, but, where possible, uses basic data structures specified by the Mica I/O architecture instead of specific data structures internal to the RR function processor.

The VMS/RR database contains a single port block for each client or server process that opens a port to RR. In the VMS implementation, each process is limited to having at most one RR port. Information contained in the VMS/RR *port block* is maintained by Mica/RR in a Mica I/O channel object object, created by the DFS client and server.

VMS/RR establishes one DECnet logical link for each remote node with which it exchanges messages. For each link a *connection block* is created and maintained in a linked list within its internal database. Linked lists of *session blocks*, corresponding to sessions opened through any of the ports, are attached to the connection block that represents the logical link over which transactions within the context of the session are communicated. Mica/RR uses the channel objects corresponding to DECnet connections as connection blocks, and maintains internal lists of session blocks pointing to them.

The *transaction blocks* used in VMS/RR correspond to Mica I/O request packets (IRPs). The request-dependent portion of the IRP contains those fields associated with transaction blocks in VMS/RR necessary for managing requests and responses: for queuing requests to channels, for maintaining the transaction information while it is being processed by the RR FPU and subsequent layers, and for returning the request completion to the user process or FPU.

45.1.1.4.2 Design Considerations and Issues

There are two main design considerations in RR: elimination of data copying as data moves from the lowest layers of DECnet up through RR to the calling FPU or process, and minimizing context switches as control of the data moves between the layers. The support that RR supplies for servers is somewhat different than that supplied to clients. Clients simply make requests, via Mica I/O channels, and receive completions through an AST or event. Servers, on the other hand, must be prepared to receive requests from remote clients, the number and size of which it cannot know beforehand.

On transmit, data copying is avoided by providing a mechanism for messages to be constructed in the user's address space, for that message to be locked in memory, and for a pointer to the message to then be passed through DECnet to the NI controller. Because messages are built dynamically using RCL in-line procedures, the user is required to know beforehand the message format and the sizes of its various components, to allocate a region of the appropriate size for building the message, and to construct it contiguously within this region. Otherwise, due to fundamental NI controller limitations on the number of discontiguous regions that can be combined to form a single message, the message needs to be copied at least once.

On receive, at least one data copy is necessary, because many users are multiplexed over logical links by RR. RR receives the messages as a linked list of ordered fragments from DECnet, and then copies them into a user-provided buffer. Additional copies may be required by the user to position the received data correctly within its address space. If buffers have not been provided ahead of time by the users, then RR holds on to the incoming messages, notifying the user that it needs to provide buffer space for the incoming message, until either the buffers are provided, or the message is timed out and discarded.

Much context switching will be avoided by allowing the user thread to pass through RR and continue down through DECnet. This will, in fact, be necessary to avoid data copying. Similarly, DECnet threads will be able to execute the RR code necessary to complete a request through a callback mechanism.

The detailed design for eliminating both data copying and context switching depends on the specification of the DECnet interface, and is thoroughly described in the chapter body.

45.1.2 RCL

RCL can be thought of as a higher-level protocol layered on top of RR to provide a general RPC facility. RCL is responsible, on the client side, for packaging procedure calls to a remote service into formatted messages, sending them through the RR transport, and unpackaging the formatted response messages returned by RR, and for the corresponding functionality on the server side.

Although logically a separate layer, RCL is implemented in VMS as a set of subroutines and macros. DFS uses these to format remote file requests and to pass the requests to the RR device driver. For efficiency of design and execution, Mica/RCL uses a similar approach. RCL is implemented as a system subroutine library.

45.1.2.1 Structure of RCL Messages

The RCL message format is based on expressing procedure calls and responses to procedure calls as lists constructed from a small set of fundamental elements:

- Program

  Each RCL message contains one and only one program element. It always occurs first in the message, and is used either to describe the overall contents of the message in the case of a request, or to return response data. Unfortunately, "program" is a rather misleading name for this data element. It is simply the highest-level descriptor within an RCL message, describing how many of the next lowest level elements, called "functions", are present in the message, and an overall status or result for the message. Its use and meaning are entirely application dependent.

- Function

  The basic entity transported in RCL messages are functions. Functions are simply groupings of element descriptors that contain actual data. They represent an organization level below programs, but above the remaining three types of elements. Their meaning and use are again entirely application-dependent. Messages generally include one or more functions, each of which

consists of a leading function element, followed by a list of associated elements which describe the arguments to the function. These associated elements include:

- Integer

  A simple data element describing a 32-bit signed integer

- Vector

  A data element used for describing logically contiguous collections of bytes; interpretation of the data is left to the higher-level software using RCL

- Nil

  A placeholder element for marking the absence of arguments

45.1.2.2 Subroutines

The RCL subroutine library includes the following types of subroutines:

- Message Building Subroutines

  Included in this category are the necessary subroutines for constructing RCL element lists. For example, there are separate routines for adding function elements, adding integer elements, and so on. Also included are subroutines for allocating and initializing RCL messages.

- Message Unpacking Subroutines

  Corresponding to the message building subroutines are the routines for unpacking the RCL messages on the other end. Routines in this category include those for getting the server-supplied status, for getting the next element within the message, and for copying vector data into user-defined memory locations.

- Communication Subroutines

  These are a set of subroutines for sending RCL messages. Included are subroutines for sending messages on the client and server side and subroutines for opening and closing sessions.

CHAPTER 46

DISTRIBUTED FILE SERVICE CLIENT FUNCTION PROCESSOR

46.1 Overview

This overview summarizes the design and function of the Mica DFS client function processor (Mica DFS client), which allows a Mica system to access files on remote Mica or VMS systems.

46.1.1 Requirements

The Mica DFS client satisfies the following requirements:

- To provide a transparent interface from a Mica system to files on a remote Mica or VMS system
- To translate Mica I/O requests into the protocol understood by other DFS implementations
- To be controlled through Mica system management

Transparent Interface. To provide a transparent interface from a Mica system to remote files, the Mica DFS client implements the Mica disk-file function processor interface class (see Chapter 24, Disk File System Function Processors). The Mica DFS client, however, supports only some of the I/O functions supported by the disk-file function processor class. For a list of the functions that the Mica DFS client supports, see Section 46.1.2.

Mica-to-DFS Translation. The Mica DFS client translates Mica I/O requests into the protocol understood by the other DFS components. Currently, this protocol is an RPC implementation of the QIO interface to the extended QIO processor of the VMS file system. The RPC mechanism is the request/response transaction protocol (RR). For a description of protocol that DFS uses, see the DFS File Protocol Document.

Management. Configuration of the Mica DFS client is controlled by Mica system management, using general MOUNT facilities. For details, see Chapter 32, System Management, and Chapter 43, Distributed File Service Introduction.

46.1.2 Functional Interface

The Mica DFS client is an I/O function processor; it supports the common function-processor entry points documented in Chapter 8, I/O Architecture.

Of the functions supported by the Mica disk-file function processor, the Mica DFS client supports only the following ones:

- FPU Functions
  - *io$c_get_fpu_info*
  - *io$c_ready_fpu*
  - *io$c_unready_fpu*

- Directory Functions
  - *io$c_dfile_read_dir_entries*
  - *io$c_dfile_modify_dir_entries*
- Access, Creation and Deaccess Functions
  - *io$c_dfile_access*
  - *io$c_dfile_create*
  - *io$c_dfile_deaccess*
- Attribute Functions
  - *io$c_dfile_read_attributes*
  - *io$c_dfile_write_attributes*
- File Storage Functions
  - *io$c_dfile_allocate_storage*
  - *io$c_dfile_deallocate_storage*
- Data Transfer Functions
  - *io$c_dfile_read_file_data*
  - *io$c_dfile_security_erase*
  - *io$c_dfile_write_file_data*
- Memory-Management Support Functions
  - *io$c_dfile_mmclone_access*
  - *io$c_page_read*
  - *io$c_page_write*

The following function codes, though supported by the Mica disk-file function processor, are not supported by the Mica DFS client.

- *io$c_item_nonpaged*

46.1.3 Internal Design

The Mica DFS client operates on requests sent as arguments to the Mica *request_io* system service. The Mica DFS client first captures these arguments, and checks them for validity. It then translates each Mica file-I/O request into the DFS file-protocol format understood by the remote DFS server.

Multiple Operations

Some operations that can be expressed in a single Mica I/O request translate into several intermediate DFS I/O requests. To process these intermediate requests, the Mica DFS client uses one system thread from a pool of such threads. This thread processes each request synchronously—that is, the thread awaits the completion of each request before processing the next. This synchronous processing continues either until an error occurs or until all the operations complete successfully.

Error Reporting

If an error occurs on the remote server system during execution of a DFS file-protocol function, the server returns error information to the Mica DFS client as a VMS-formatted I/O status block and status. Then, before the I/O completes, the Mica DFS client translates this returned information into error codes understood by the Mica disk-file function processor.

FPU Naming

DFS accesses remote disks and directories by way of *access points*, each of which translates to a DECnet node address and an access point ID. Each access point is registered with the DECnet name server.

In Mica DFS, access points are mounted by calling the Mica DFS client at its *create_fpu* entry point, supplying as a call parameter the name of the access point to be made available. To translate this name, the Mica DFS client calls the DECnet name server, then communicates with the remote server to declare the new access point available for further activity. Next, the client converts the access point name to an FPU name, which it then enters in the $BACKTRANS logical name table. This logical name table is documented in Chapter 43, Distributed File Service Introduction.

For details of the translation from an access point name to an FPU name, see the internal design of the *intialize_fpu* routine described later in this chapter.

CHAPTER 47

DISTRIBUTED FILE SERVICE SERVER

47.1 Overview

This chapter discusses the Mica Distributed File System (Mica DFS) server. For an introduction to Mica DFS in general, and the Mica DFS server in particular, see Chapter 43, Distributed File Service Introduction. Management of the Mica DFS server is discussed in Chapter 44, Distributed File Service Management.

47.1.1 Sessions

The Mica DFS server uses the *request/response function processor* (R/R FP) to communicate with its clients. The R/R FP provides RPC-style communication on top of a reliable communication service, such as DECnet.

An *R/R session* is a conversation about a particular object within the context of a service, such as Mica DFS. A session is used to represent state being stored on a server across multiple transactions. The R/R FP is responsible for fair allocation of communication resources between sessions. Chapter 45, Distributed File Service Communication Function Processor, provides an overview of sessions.

Mica DFS sessions represent the following state on the server: the authenticated identity of a user, and the channel to an open file if one is currently open. Note that it is the responsibility of the service, not R/R, to authenticate its users. Two different users, using the same instance of the Mica DFS client software, that open the same file, are assigned different sessions and different file channels.

All messages are sent to the server in the context of a session. Each user's messages need to be sent on a session on which that user has been authenticated. A session can also be associated with an open file. File operations intended for a particular open file need to be sent in the context of a session associated with the desired file. File operations which are not directed at any particular open file (for example directory lookup) can be sent on any authenticated session.

47.1.2 Server Process Implementation

The Mica DFS server is implemented as a multithreaded, user-mode process. A multithreaded design is desirable because it increases concurrency that can be taken advantage of by multiprocessor hardware and disk controllers. The number of threads used by the server process is adjusted automatically. The server is implemented in user-mode for reasons of reliability:

- A failure in a user-mode server is less likely to affect the rest of the system.
- A user-mode server can take advantage of user-mode impersonation services, which provide for the changing of identities in a controlled manner.

All server threads share one channel to the R/R FP. When ready to receive a file request, each thread issues a synchronous read to the R/R FP. The R/R FP completes the reads in order of arrival when messages are available. If all Mica DFS server threads are executing requests, no additional requests can be processed until a thread becomes free. The R/R FP has responsibility for flow control of unread messages.

Any server thread is capable of processing any request on any session. Session state (that is client identity and file channel) is shared between threads. Because of the multithreaded nature of the DFS server, the order in which requests are processed may be different from the order in which they were received, even for requests on the same session.

47.1.3 File Protocol

The file protocol implemented by the Mica DFS server is a RPC implementation of the VMS file system XQP QIO interface. Translation is required to execute the VMS-style requests using the Mica disk file system function processor. The difficulty of translation is relatively small since the Mica file interface is an evolution of the VMS interface on which the file protocol is based.

The file protocol expects the file server to do wildcard processing. The Mica file system does not handle wildcards and leaves this function to higher layers. The Mica DFS server therefore includes the capability to resolve wildcards.

Both the Mica and VMS file systems depend on a lock manager to coordinate access to parts of a file. DFS users on different nodes are in different lockspaces and will not be informed of conflicting access to the same file. Mica DFS prevents any inconsistencies by converting opens allowing write sharing to exclusive access. A mode may be specified in the file protocol for the open operation to cause the server to relax this restriction.

47.1.4 Security

As discussed in Section 47.1.1, DFS, not the R/R communication layer, is responsible for authentication of users. When a new session is to be authenticated for a particular user, DFS sends a special *set_environment* message to the server providing the user's node and username. The Mica DFS server then uses Mica security services to map the user's remote identity into a local security profile. Chapter 10, Security and Privileges, discusses Mica security in more detail.

When a Mica DFS server thread receives a request, it must assume the security profile associated with that session before issuing any Mica file operations.

47.1.5 Caching

The disk file system function processor will not be implementing a disk block cache for first release. The Mica DFS server, whose function is closely related to this function processor, will follow suit and not implement one either. The reasons for this are as follows:

- Caching is enhanced functionality which is not required for first release.
- Waiting until we can analyze a working system will give us greater insight into the need for caching.
- For efficient use of memory and reduced mechanism, the Mica DFS server should utilize the disk block cache provided by the disk file system function processor instead of its own.
- Mica has support in other parts of the system to reduce disk accesses, specifically file system structure caching, image caching, and RMS readahead.

47.1.6 Buffering

Each Mica DFS server thread will own one large, static buffer allocated in the server's address space. That buffer will be used to receive requests from R/R. After the request has been processed, the buffer is used to form the reply. Tranfers to/from the disk file system function processor will be made directly from/to this buffer.

47.1.7 Accounting and Quota Enforcement

An accounting record answering for aggregate server resource usage will be generated when the Mica DFS server exits. \It has not been decided if accounting records will also be generated to answer for per-client server thread resource usage.\

The Mica DFS server is subject to process-wide quotas on resources such as executive pool. Enforcement of such quotas by Mica will prevent the server itself from acting unfairly.

Per-client quotas apply to high level resources (for example files) and are enforced by the providers of those resources. The Mica DFS server does not provide any resources itself, and thus enforces no quotas itself. The disk file system function processor is responsible for enforcing any file-related quotas, for example disk block usage.

47.1.8 Failure Recovery

The Mica DFS server expects the R/R FP to provide notification of session termination in the event of a client failure. These sessions will be removed and will no longer be recognized. The files associated with these sessions will be closed.

If the Mica DFS server crashes and restarts, all files open on any client are no longer recognized. Access points previously mounted by a client system are transparently remounted on use.

Database Server

This set of chapters describes the components of Mica that provide database server support.

CHAPTER 48

CHEYENNE OVERVIEW

48.1 Overview

Cheyenne provides customers with highly available, very fast, relational database capability. Its target applications mostly involve on-line transaction processing (OLTP) environments. Cheyenne implements the DIGITAL Database Architecture (*DDA*), allowing the growing body of applications built using DIGITAL's Rdb product set to access Cheyenne databases with little or no change.

Cheyenne is unique from other DIGITAL DDA products in that it has both hardware and software components. Current DIGITAL database products run as privileged application programs on the same system as the users' application code. Cheyenne consists of one or more PRISM systems running the Mica operating system. Only one application runs on the PRISM systems: the Quartz relational database system. User application programs run on client systems that communicate with the database server over a network connection (NI or Ethernet).

Most Cheyenne systems are used in transaction processing applications. These include financial applications (for example, funds transfer, bank machines, stock transactions), reservation systems (for example, hotel reservations, sporting events), inventory management applications, and so on. Such transaction systems consist largely of two types of application transactions:

- Well-defined, short query-and-update transactions
- Long, background, read-only report generation or roll-up update transactions

The transaction processing monitor and application programs run on the client systems, not the database server. Cheyenne provides only the database services to these applications.

Cheyenne systems are available in two basic configurations:

- Standard
- Highly available

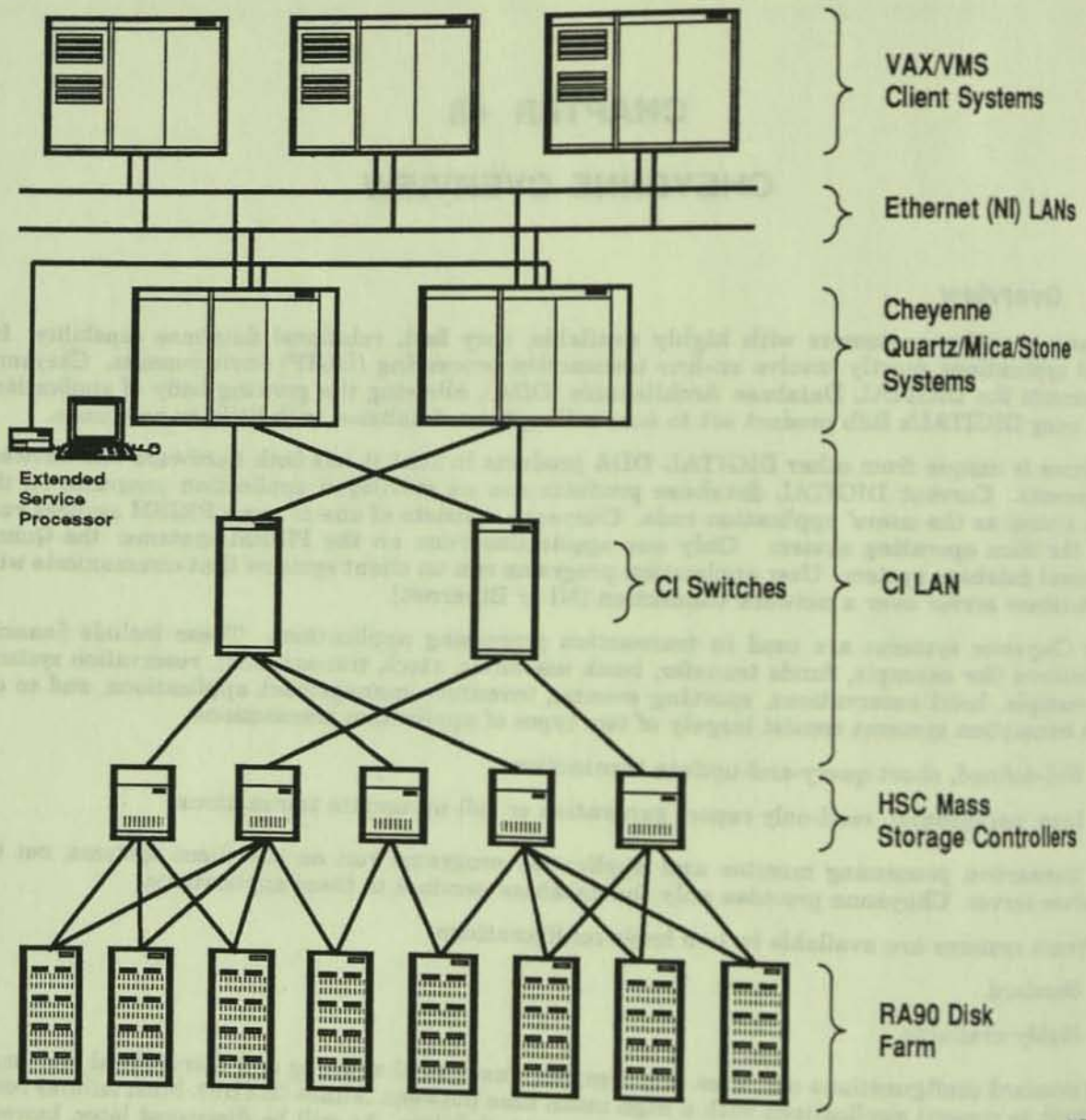
The standard configurations use Mica features, such as disk shadowing and hierarchical fault management, to present applications with a high mean time between failure (MTBF). Most failures result in performance degradation, instead of service unavailability. As will be discussed later, however, there are single points of failure that result in the database becoming unavailable to applications.

The highly available Cheyenne configurations tie together multiple Quartz/Mica/Stone systems to form a database server that is considerably more fault-tolerant than the standard configurations. Highly available systems are impervious to single points of failure. No single failure (with the exception of power failure to all constituent systems) can cause the database to be unavailable to its applications. The performance may degrade, but the database remains available. Likewise, most multiple-point failures only degrade performance.

The highly available Cheyenne configurations are provided as tools for building highly available applications. Other necessary tools include highly available front-end systems and communications components. Cheyenne can be combined with these tools to produce complete application systems that are impervious to single points of failure.

Figure 48–1 shows an example of a highly available Cheyenne and client system configuration.

Figure 48–1: Highly Available Cheyenne Configuration



48.1.1 Product Goals

The priority of Cheyenne product goals is uncommon for a DIGITAL product:

1. 100% data integrity
2. High reliability and availability
3. High performance

This set of goals presents unique challenges for testing and verifying the product. Although we recognize that these challenges exist, it will be some time before they are resolved.

48.1.1.1 Data Integrity

Cheyenne's primary goal is to handle user data without corrupting it. This implies that all errors must be detected and dealt with appropriately. These include disk errors, channel and bus errors, controller errors, memory errors, data path errors, and failures of computational elements. It is the hardware's responsibility to detect and report such errors; Mica records every error in an error log. Mica then attempts to correct the error transparently, for example through the use of a counterpart in a disk shadow set. Errors that cannot be corrected are reported to and handled by Quartz, the database system software.

48.1.1.2 Reliability and Availability

The second product goal is to provide a highly reliable, highly available system. High reliability means that the system seldom fails to operate. High availability means that the data managed by the server is rarely unavailable to client systems. All Cheyenne configurations are highly reliable; customers can elect to purchase highly available configurations.

High reliability is achieved by detecting, recovering, and logging every hardware error. Error recovery may be performed by the hardware, by Mica, or may require the intervention of Quartz. Some failures require the intervention of front-end software to resubmit transactions that were aborted because of the error. A Mica process monitors the error log to detect patterns. If a failing pattern is discovered, the failing component is automatically removed from the active system configuration and redundant components take over the load. The result may be a degradation of performance, but the database remains available to the application. Failures of unreplicated components may result in the data becoming unavailable. Service personnel eventually replace components that are moved out of the configuration.

High availability is achieved by replicating hardware components within Cheyenne. The load is shared by the components until one fails. The entire load is then taken up by the remaining components. Thus, replicated hardware provides Cheyenne with high availability *and* high performance. For example, two or more Quartz/Mica/Stone systems can be bound together to form a highly available database server. Both are busy servicing the transaction load. If any one Quartz/Mica/Stone system fails completely, the remaining systems pick up the load. The database remains available to applications as long as at least one constituent Quartz/Mica/Stone system remains operational and is configured to access the disks on which the database resides. Mass storage and intersystem communications components can also be replicated.

High availability is achieved by binding together two or more Quartz/Mica/Stone systems to form a highly available database server. If any one Quartz/Mica/Stone system fails completely, the remaining systems pick up the load. The database remains available to applications as long as at least one constituent Quartz/Mica/Stone system remains operational and is configured to access the disks on which the database resides.

48.1.1.3 Performance

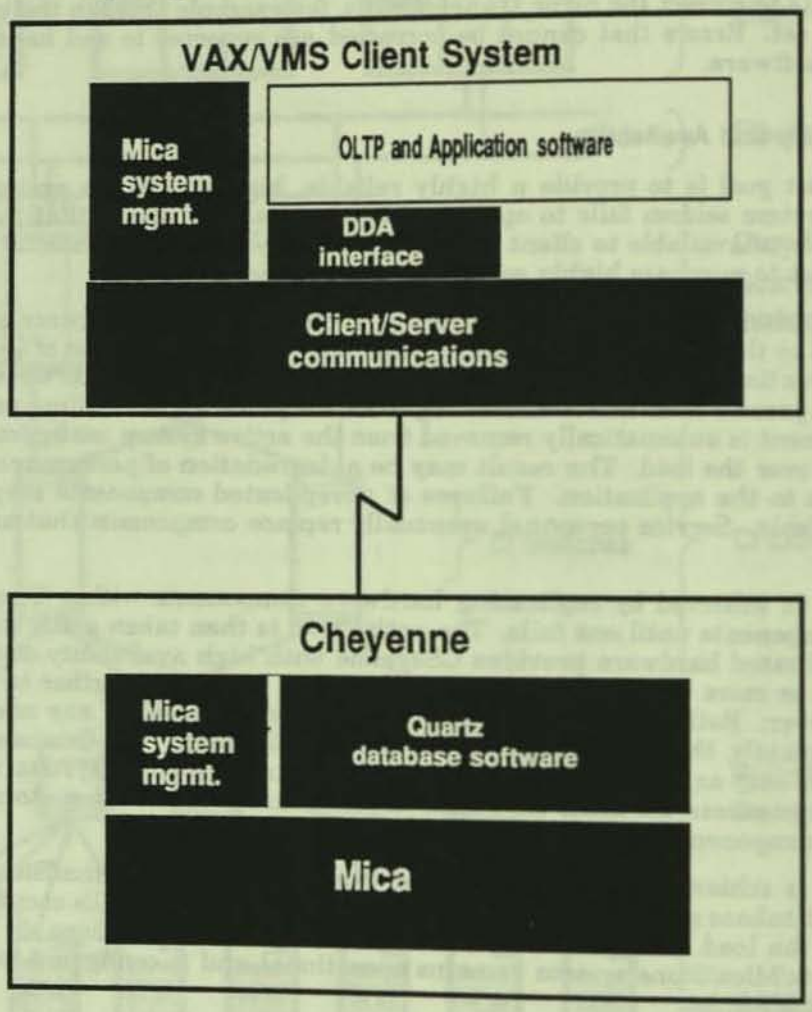
Cheyenne systems come in a variety of performance ranges. The lowest performance system is designed to run 100 industry-standard debit/credit transactions/second. The highest performance configuration can run over 600 of these transactions each second. The database software is also designed to handle large, read-only transactions efficiently.

High performance is achieved through the use of sophisticated database manipulation algorithms, parallel processing, large amounts of physical memory, disk shadow sets, extensive data caching, and careful design of Mica and database interfaces. The software is designed to take advantage of the large number of processors and enormous physical memory provided by Stone.

48.1.2 Components

The following sections enumerate Cheyenne's many components. Figure 48–2 shows how the components fit together (Cheyenne components are highlighted).

Figure 48–2: Software Layering



48.1.2.1 Stone

Cheyenne is built with Stone systems, which implement the 32-bit PRISM architecture. A Cheyenne database server consists of one or more Stone systems, mass storage, and an optional extended service processor. Each Stone system has the following components:

- 2 to 8 scalar processors
- 64 Mbytes to 1 Gbyte of main memory
- 1 or 2 XMI I/O busses
- Assorted intersystem communications adapters
- Wildcat disk and tape controllers (optional)

Each Cheyenne supports up to 1 Tbyte of disk storage, thus allowing a Cheyenne to manage up to 256 Gbytes of user data.

48.1.2.2 Extended Service Processor

The Stone systems share an optional extended service processor. The extended service processor is a VAXstation running VMS, linked to the Stone systems through a private Ethernet network. The extended service processor provides comprehensive symptom-directed diagnostic services as well as access to the system management interface.

48.1.2.3 Mass Storage

Cheyenne supports both DSA-1 and DSA-2 disks. DSA-1 disks (RA8x, RA70, and RA90) can be accessed through Wildcat (HSX) disk controllers. They can also be connected to the system through HSC mass storage controllers attached to the Stone systems through CI (XCA) adapters. Each DSA-2 disk includes its own controller. DSA-2 disks are attached to the Stone systems through CI adapters.

Removable media is provided through one of the following (a choice has not yet been made):

- TA90 cartridge tape, from either the Wildcat or HSC controllers
- A variant of the TA90, from an as-yet undefined XMI controller
- Ptolemy write-once optical media

48.1.2.4 Mica

The Mica operating system runs on the Stone systems. Cheyenne uses the following Mica features extensively:

Component	Special Features Required by Cheyenne
Kernel	Symmetric multiprocessing (SMP)
Executive	Common logging
I/O subsystem	Disk and tape function processors
	ODS-2+ disk volume support
	Disk shadowing and striping
Intersystem communications	DECnet
	Remote procedure calls
	Client/server
	Interprocess communications between Stone systems
	CI and NI support
System management	Configuration management
	Network management
	Security
	On-line diagnostics

48.1.2.5 Quartz

Cheyenne's relational database system is Quartz, developed by Storage Systems in Colorado Springs. Quartz is a DDA-compliant database manager. Although Quartz is optimized for use by on-line transaction processing applications, Quartz also contains many features that make it attractive for end-user information management.

48.1.2.6 Client Software

Portions of the Cheyenne product are software running on VMS client systems. The following sections enumerate these components.

48.1.2.6.1 Communications

The client side of the client/server communications runs on client systems. There are three components to the client/server communications:

- Database communications layer
- Reliable communications service (RCS)
- DECnet

48.1.2.6.2 Mica and Quartz System Management and Database Administration

The system management requirements of the Quartz/Mica/Stone systems that make up a Cheyenne system include software installation, network configuration, system security, mass storage configuration, running on-line diagnostics, and mass storage backup. All of these functions can be accomplished remotely from a properly authorized client system. They can also be performed from the Stone console subsystem.

The Cheyenne database administration functions include database creation, tuning, security, management, and backup; allocation of resources to various databases; and resource usage accounting. All of these functions can be accomplished remotely from a properly authorized client system. They cannot be performed from the Stone console subsystem.

48.1.2.6.3 Security

Cheyenne resources and databases are protected from unauthorized access through a hierarchical authorization scheme. The layers include:

- DECnet-Mica

  This layer validates connections based on the originating system and user name. Note, however, that this is the user name for the process that sets up the connection, which is usually not the process making the DDA request. Mica's security scheme allows the system manager to restrict network access based on system identification and user name.

- Reliable communications service

  This layer provides the Quartz database management software with information it needs to identify processes that issue database requests.

- Database communications layer

  The database communications layer translates DDA procedure calls into messages sent to the RCS layer. No authentication is performed by this layer.

- Quartz database management software

  The database software uses the identity provided by the RCS layer to validate the intended access. This validation is in two steps:

  — Database file protection

    Quartz identifies the accessor to Mica and asks Mica if the access is to be allowed.

  — DDA-defined database protection

Quartz uses security information placed in the database by the database administrator to determine if the intended access is to be allowed.

48.1.2.6.4 Database Tools

The database access method (Quartz) is only one component of a modern database management system. Other components include:

- Database definition tools
- Database tuning tools
- Database analysis, backup, and repair tools
- Program development environment
- Interactive query package

Some of these tools are generic to any DDA offering (for example, program development environment, interactive query package, and possibly the database definition tools), while others are unique to Cheyenne (the database analysis and repair tools, and some of the tuning tools).

48.2 Target Customer Base

Cheyenne's primary target market is on-line transaction processing (OLTP) applications. OLTP applications are characterized by relatively short, well-defined transactions. Although the number of transaction types is small, fantastically large numbers of transaction instances are run. For example, the high-end Cheyenne configurations are targeted towards performing 600 qualified Debit/Credit transactions/second (see the *CRDK Tutorial* for a description of a qualified Debit/Credit transaction). A large proportion of OLTP transactions update the database.

Cheyenne is also targeted towards transactions that up to now have been referred to as *ad-hoc*. Ad-hoc transactions are usually not repeated, and may not be defined until the transaction is run. Most ad-hoc transactions read the database, but do not update it. Ad-hoc transactions may require scanning or accessing significant portions of the entire database. Ad-hoc applications are also referred to as end-user information management (EUIM).

Cheyenne includes many features that enable it to address these target markets:

- High availability

  Current DIGITAL systems are not highly available. Cheyenne is specifically designed to provide highly available access to databases. Cheyenne hardware and software failures affect only running transactions; new transactions can be started within two minutes of a failure. Self-contained transactions can be automatically resubmitted by front-end software, thus masking the failure from applications and application users.

  High availability is critical to many transaction processing applications. Customers demand 100% uptime, 7 days per week, 24 hours per day; anything less would make it impossible for the customer to use Cheyenne to run their applications.

- High throughput

  Cheyenne has considerably more processing power and I/O bandwidth dedicated to database processing than DIGITAL layered database products. Quartz is specifically tailored towards handling the types of transactions prevalent in OLTP applications. As a consequence, Cheyenne can process a greater number of transactions in a given period of time than other DIGITAL database products.

- Faster transactions

  Quartz is designed to process relatively short update transactions very quickly. This is extremely critical for many OLTP applications. At the same time, Quartz is also designed to handle large queries efficiently. Quartz takes advantage of the large number of processors, enormous memory capacity, and high I/O bandwidth to decompose large queries and execute the pieces in parallel. This makes it possible to execute large transactions in reasonable amounts of time. Cheyenne is fast enough to make practical queries that would not be possible in smaller systems. This feature is essential for ad-hoc queries on large databases.

- Bigger databases

  Cheyenne is designed to support databases up to 256 Gbytes in size (this requires about one Tbyte of mass storage, due to Quartz overhead and data-shadowing requirements). Quartz can not only query such databases, but can load, backup, and reorganize them in reasonable amounts of time (less than a single eight-hour shift). This capability makes it practical to implement large-scale applications on top of Cheyenne.

These application classes describe the types of applications that run on Cheyenne, but they do not describe the people who interact with Cheyenne. It is useful to consider how various classes of users view Cheyenne. We can use these perspectives to tune interfaces for the people we expect to use them. The following sections describe the ways in which various classes of users interact with Cheyenne.

48.2.1 Application Users

Cheyenne makes it possible for DIGITAL to reach new classes of applications. These are primarily commercial, transaction-processing applications, in areas such as as banking, brokerages, inventory control, and personnel. Users of these applications may never know that Cheyenne is there, but they will notice that their work proceeds faster, with fewer interruptions. This allows more work to be done in a given amount of time than is possible using DIGITAL layered products instead of Cheyenne.

48.2.2 Application Writers

Writers of application programs may develop their programs on top of existing DIGITAL layered products (for example, Rdb/VMS). The database accessed by a program may be managed initially by a layered product, then later migrated to Cheyenne. Programs should require no changes to access the migrated database, with the possible exception of the database name (logical names can be used to mask even this small change). Alternatively, application programs can be developed directly on top of Cheyenne databases. Migrated applications may need to be recompiled and relinked before they can access their database on the Cheyenne system.

In short, application programmers, like application users, may be unaware of when, or if, Cheyenne is being used.

48.2.3 Database Administrators

Database administrators belong to the class of users most affected by Cheyenne. The database administrator is responsible for designing the database, defining its structure and layout, choosing a database manager, and for tuning the database to meet the demands of application users and writers. Although Cheyenne databases are DDA-compliant, many of the database layout and tuning choices are unique to Cheyenne.

Database administrators manage the following aspects of a Cheyenne system:

- The number and power of the Stone systems in the Cheyenne server
- The number and type of disks
- The partitioning of databases among disks and Stone systems

- The layout of relations and choice of fast access structures for each database (tuning the databases)
- The use of security to limit unauthorized access to Cheyenne databases
- Day-to-day management of the database, including backup
- Repair of broken databases

Database administrators have additional duties not included in this list. These include logical database design, a function common to all DDA databases, not just Cheyenne databases.

In summary, database administrators are concerned with the configuration of a Cheyenne system and with how these resources are distributed among databases.

48.2.4 System Managers

The role of the Cheyenne system manager overlaps with that of the database administrator. System managers are oriented even more towards dealing with the physical resources in the Cheyenne configuration. System managers concern themselves with the following:

- Client/server network management
- Binding multiple Stone systems together to form a highly available Cheyenne system
- Intra-Cheyenne (inter-Stone) communications management
- Assignment of physical mass storage to various uses (this clearly overlaps with the database administrator role)
- Accounting for resource usage
- Taking corrective actions when components fail or are about to fail
- Physically configuring the system

One useful distinction between system management and database administration is that database administrators manage the static and long-term allocation of system resources, while system managers concern themselves with short-term control of resources.

48.2.5 Operations Staff

Operators are responsible for the day-to-day operation of Cheyenne, with duties defined by the system manager and database administrator. Generally, operators perform backups and manage the media containing long-term journals. They also interact with Cheyenne following a device or system failover; for example, operators may implement policy decisions regarding when to remove failing components (such as disks) from the system.

48.2.6 Software Support Personnel

Software support personnel are the first resources a customer calls upon for knowledge of Cheyenne. The software support specialist needs extensive knowledge of how the system will be used. This allows the specialist to advise and train the customer's database administrators and programmers.

Software support personnel are also the first line of defense when the Cheyenne system fails and no hardware cause can be found. They attempt to identify the conditions that cause the failure and determine why the failure is taking place. This requires extensive knowledge of the internals of the system. It also requires that tools be available to analyze a running system as well as a failed system. Wherever possible, the specialist helps the customer work around the problem. The specialist needs thorough knowledge about the on-disk structure of the database to use the repair utilities to fix a damaged database.

The specialist works with engineering to resolve especially difficult problems. Once the problem has been identified, the specialist may apply emergency software updates provided by engineering. It is extremely unlikely that software specialists will patch, modify, or customize Cheyenne software.

In summary, software support personnel need extensive knowledge about the internals of the implementation, and details about the on-disk structure. They need tools to analyze the system, and utilities to probe and repair damaged database files. Finally, they need a means of applying emergency software updates.

48.2.7 Hardware Service Personnel

Hardware service personnel install and repair systems. They need to be aware of the issues affecting high availability (see Section 48.5.1).

The primary distinction between Glacier and high availability Cheyenne configurations is the number of Stone systems each may include. Standard Cheyenne configurations are essentially identical to Glacier systems, differing only in that Glacier processor modules can have scalar/vector processors, while Cheyenne uses scalar/scalar processor modules. Service personnel will view the two systems as very similar, and practical considerations suggest that the same people will service both Glacier and Cheyenne systems. Thus, it is unlikely that service personnel will be aware they are dealing with a database machine. For highly available Cheyenne systems, however, service personnel must be aware of how to service portions of the Cheyenne system without affecting overall system availability.

48.3 Hardware Components

The following sections briefly describe the hardware components used by Cheyenne.

48.3.1 Client Systems

To be a Cheyenne client, a computer system must implement DECnet, and have client-resident DDA communications software. For the initial release of Cheyenne, only VMS systems meet these requirements. Future releases of client-resident software may enable ULTRIX systems to be clients.

Although IBM systems cannot connect directly to Cheyenne, they can become Cheyenne clients by using layered DDA products (for example, *VIDA*) that are connected to Cheyenne through VMS.

48.3.2 Standard Configurations

The standard Cheyenne configuration consists of a single Stone system, optional extended service processor, mass storage, and client communications hardware.

The Stone system consists of from one to four scalar/scalar PRISM processor pairs, memory, I/O adapters and busses, power and packaging, and a console subsystem.

The mass storage for a standard Cheyenne configuration is described in Section 48.3.4.

Client systems communicate with Cheyenne using DECnet and one or more network interconnects (NI or Ethernet). Additional NIs provide additional throughput and improve availability. Future versions of Cheyenne will allow clients to be attached to Cheyenne using the CI bus.

48.3.3 Highly Available Configurations

A highly available Cheyenne configuration is built from two or more Stone systems, an optional extended service processor, mass storage, and client communications hardware.

The Stone systems are identical to those used for the standard configurations. The Stone systems within a Cheyenne are tied together using the CI bus. The Cheyenne workload is shared among the Stone systems during normal operation. If a Stone system fails, its workload is reassigned to the remaining Stone systems within the Cheyenne; client systems see a momentary glitch, and many outstanding transactions are aborted. The client systems can immediately resubmit their transactions, usually transparently to both the transaction processing monitor and users.

The mass storage for highly available Cheyenne configurations is described in Section 48.3.4.

The client communications hardware for highly available Cheyenne systems is identical to that for standard configurations. All of the Stone systems that make up a Cheyenne must be linked to each client system.

48.3.4 Mass Storage

The initial release of Cheyenne uses DSA-1 disks and tapes for mass storage. All DSA-1 disks are supported, although we expect that only RA70 and RA90 disks actually will be used. The TA90 tape drive is the only tape drive that we expect will be used. Some systems may include optical disk drives for backup and to store long-term journals.

DSA-1 devices can be attached to the Stone systems in a Cheyenne through either the HSX (Wildcat) controller or the HSC controller. The HSX and HSC controllers can be used for system disks in all configurations. Database and logging devices can be attached through the HSX only in standard configurations; highly available configurations require the HSC.

The CI busses used for Cheyenne mass storage must be used only to connect Stone systems and HSC controllers; no VAX may be attached to a Cheyenne CI bus. In addition, only Stone systems that are bound into the same Cheyenne system may share a CI. Further, all Stone systems in a Cheyenne system *must* be connected to every CI that is used for mass storage, so that all CI-based mass storage is visible to all the Stone systems in the Cheyenne system.

48.4 Software Components

The following sections describe the software components used by Cheyenne. Some components execute on client systems, but most execute on the Stone systems that make up the Cheyenne system.

48.4.1 Components on Client Systems

Cheyenne components execute on client systems to provide access to the Cheyenne system. These components provide several functions:

- Access to Cheyenne databases through DDA
- Cheyenne system management, network management, and database administration
- Cheyenne diagnosis and maintenance

These components are packaged as one or more layered product kits for the client operating system. We expect to provide the following layered product kits at FRS:

- VMS full client

  This kit includes all three functions, packaged for a VMS client.

- VMS run-time client

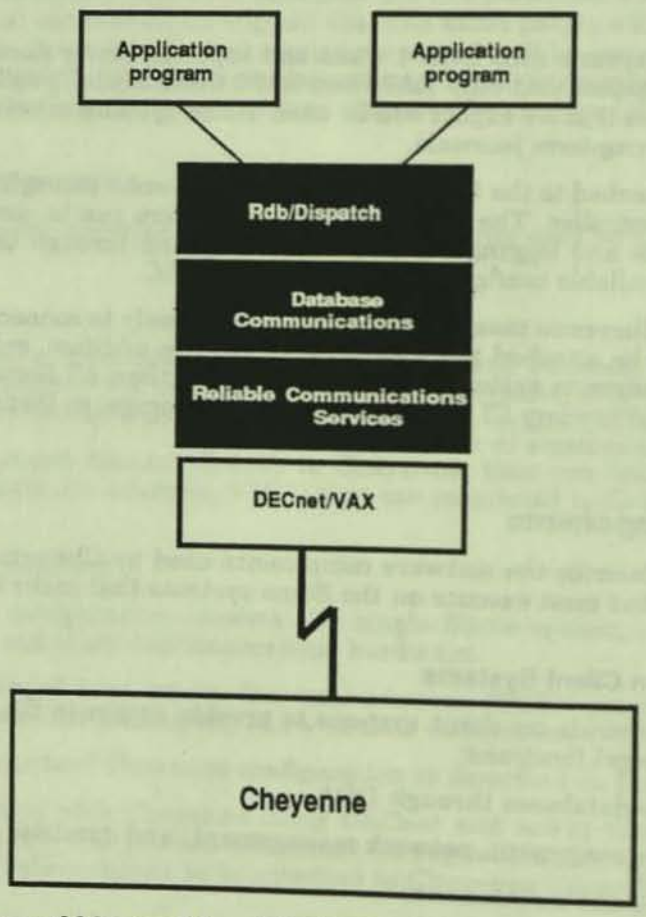
  This kit includes only the ability to access Cheyenne databases from a VMS client system.

48.4.1.1 Access to Cheyenne Databases

This function provides application users the ability to run programs that access databases on a Cheyenne system. It includes the client DDA communications modules. It could also include a number of end-user query tools (for example, an interactive SQL query processor or query-by-forms utility), but these are more likely to be packaged as layered products.

Figure 48–3 shows the components necessary for an application program to communicate with a Cheyenne system; client-resident Cheyenne components are highlighted. *Rdb / Dispatch* allows DDA calls to be directed to any DDA-compliant database manager. The *database communications (DBC)* layer provides the ability to remotely access a database. The *reliable communications service (RCS)* layer concentrates multiple DDA sessions over a limited number of DECnet logical links.

Figure 48–3: Client-Resident Cheyenne Communication Components

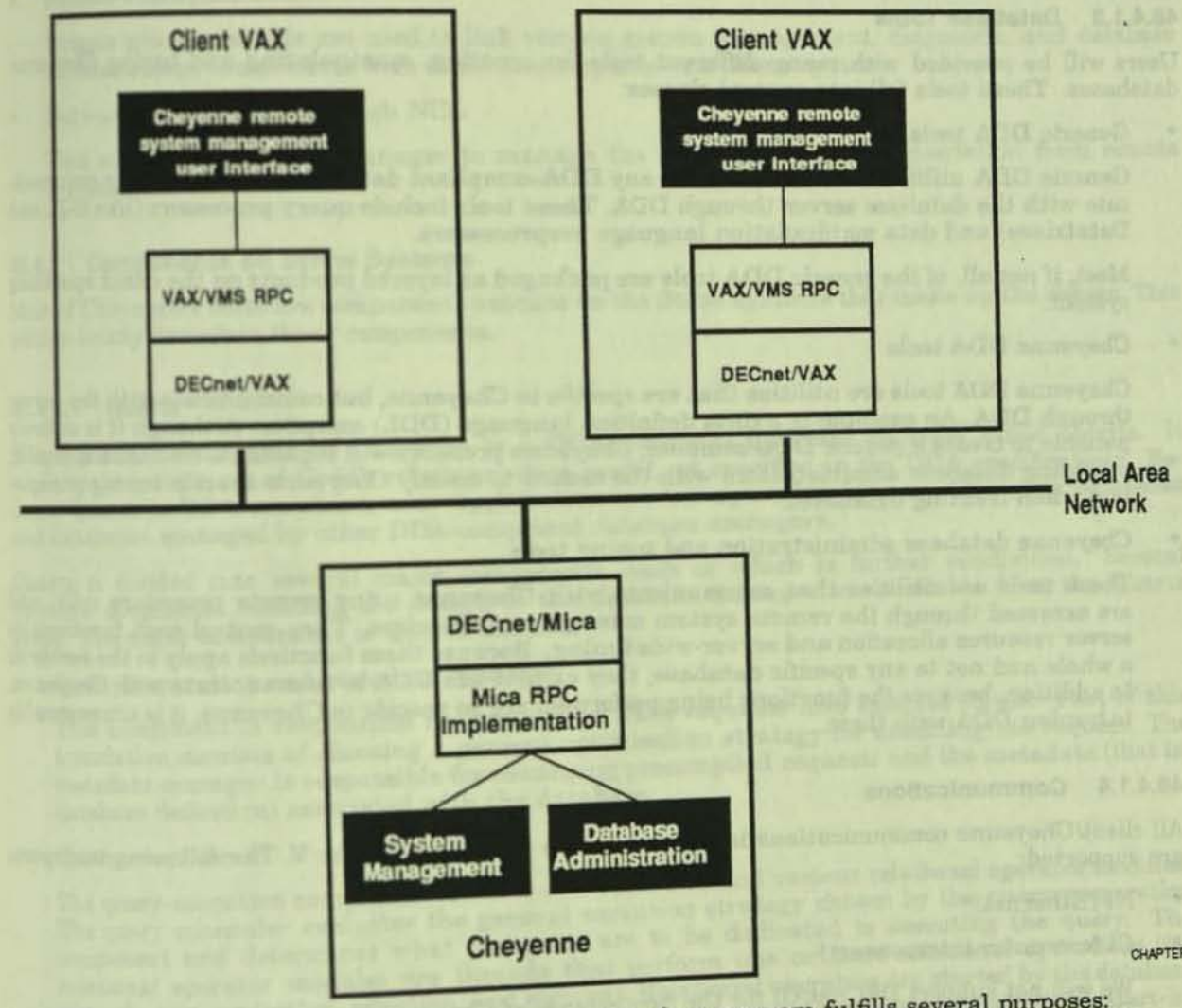


Each of these components could be used by all VMS DDA implementations, assuming that the DBC interface is not tailored specifically to Cheyenne. If these components became common, they would be packaged into a generic VMS "DDA run-time" kit. In this case, there would be no need for a Cheyenne VMS run-time client kit.

48.4.1.2 System Management and Database Administration

Because the distinction between system management and database administration is blurred in Cheyenne, it makes sense to include both functions in one package of tools. Figure 48–4 shows the various high-level components used to implement remote system management and database administration functions. System management and database administration components are highlighted.

Figure 48–4: Cheyenne Remote System Management



The Cheyenne remote system management user interface program fulfills several purposes:

- It provides the user interface to system management, network management, and database administration functions. Two interfaces are available: a command-line interface and an interactive DECwindows interface.
- It is the means by which the PRISM diagnostic monitor is invoked.

The remote system management user interface program may be implemented as several programs, tied together to act as one.

The client resident user interface uses remote procedure calls (RPC) to communicate with the system management and database administration components on Cheyenne. The Cheyenne-resident components perform the actual system management and database administration functions; the user interface directs which functions are to be performed. The user interface can also direct commands to any or all of the Stone system in a Cheyenne system. Most system management functions are directed to a specific Stone system. Database administration functions can be directed to any Stone system in the Cheyenne system.

48.4.1.3 Database Tools

Users will be provided with many different tools for creating, manipulating and tuning Cheyenne databases. These tools fall into several classes:

- Generic DDA tools

  Generic DDA utilities can be used with any DDA-compliant database manager. They communicate with the database server through DDA. These tools include query processors (like SQL and Datatrieve) and data manipulation language preprocessors.

  Most, if not all, of the generic DDA tools are packaged as layered products on the client operating system.

- Cheyenne DDA tools

  Cheyenne DDA tools are utilities that are specific to Cheyenne, but communicate with the server through DDA. An example is a data definition language (DDL) compiler. Although it is entirely possible to create a generic DDL compiler, Cheyenne probably will include its own DDL compiler, providing database administrators with the means to specify Cheyenne-specific tuning parameters when creating databases.

- Cheyenne database administration and tuning tools

  These tools are utilities that communicate with Cheyenne using remote procedure calls, and are accessed through the remote system management interface. They control such functions as server resource allocation and server-wide tuning. Because these functions apply to the server as a whole and not to any specific database, they cannot use DDA to communicate with Cheyenne. In addition, because the functions being performed are so specific to Cheyenne, it is unreasonable to burden DDA with them.

48.4.1.4 Communications

All client/Cheyenne communications is layered on top of DECnet Phase V. The following transports are supported:

- NI (Ethernet)
- CI (computer interconnect)

  We will not support DECnet/CI for the first release of Cheyenne because of the poor performance of DECnet/VAX's CI support. Because DECnet/VAX is actually faster over the NI than on the CI, there is little incentive to support DECnet/CI until performance improves.

Three forms of client/Cheyenne communication are supported:

- DDA

  This is used by client applications to access Cheyenne databases. It is also used for some forms of remote system management.

Cheyenne provides an efficient mechanism for transporting DDA requests over DECnet to the server. It is unclear at this time if this mechanism is unique to Cheyenne, or if it will become a generic DDA capability. The Cheyenne implementation multiplexes multiple DDA sessions over a relatively few DECnet logical links, eliminating the overhead of logical link creation that is present in the current remote DDA implementation, and using fewer messages to be exchanged between the client and Cheyenne than in the existing protocol. This layer is also responsible for identifying client processes to the Quartz software for security authentication.

- Remote Procedure Call

  Remote procedure calls are used to link remote system management, diagnostic, and database administration components with their counterparts on a Stone system.

- Network management through NCL

  This allows the network manager to manage the Mica DECnet implementation from remote systems.

48.4.2 Components on Stone Systems

Most of Cheyenne's software components execute on the Stone systems that make up the server. This section briefly describes those components.

48.4.2.1 Quartz

The application that runs on Cheyenne is a DDA-compliant database manager called Quartz. It implements a superset of Codd's relational data model, as specified in the DDA specifications. Because Quartz is DDA-compliant, user applications are transportable between Cheyenne databases and databases managed by other DDA-compliant database managers.

Quartz is divided into several major components, each of which is further subdivided. Several overview documents detailing the design of the database manager are available from the Quartz group. The major components of Quartz include:

- Query preparation and metadata manager

  This component is responsible for converting DDA requests into internal forms. Part of this translation consists of choosing a general optimization strategy for executing the request. The metadata manager is responsible for managing precompiled requests and the metadata (that is, database definition) associated with the database.

- Query execution

  The query-execution component consists of a scheduler and various relational operator modules. The query scheduler evaluates the general execution strategy chosen by the query-preparation component and determines what resources are to be dedicated to executing the query. The relational operator modules are threads that perform one or more relational operations (for example, join, projection, selection, and sorting). Relational operators are started by the database scheduler when it detects that a sufficient number of tuples are available to merit the start-up cost. Several relational operators can be executing in parallel, and results can be pipelined between relational operators.

- Client DDA communications

  The client communications module accepts DDA messages and routes them to the appropriate components. Most message types are passed to the query-preparation component; application data messages are always passed to the query-execution component.

Quartz has other components not mentioned here. For example, Quartz monitors the hardware configuration, because the execution scheduler needs this information. Another example is the database-administration component, which includes the ability to create databases, move them around, adjust various tuning parameters, back them up, restore them, enable afterimage journalling, disable journalling, and so on. See the Quartz design documents for details of the design of the database manager.

48.4.2.2 Mica Executive

The full range of capabilities present in the Mica executive and kernel are used in Cheyenne, with the possible exception of object security. Quartz is highly dependent on several features of Mica:

- Symmetric multiprocessing
- Multithreading within a process
- Shared memory between processes
- Hierarchical fault management architecture
- Configuration manager, error logging, and associated support throughout the executive
- Mass storage I/O, including disk shadowing
- Remote procedure call (RPC) facility
- Interprocess communications facility
- Reliable communications service

Because Cheyenne is a closed system, there is little reason to make use of Mica's object-based security. If all the interfaces into the system (that is, console and DECnet) are secure, we can take advantage of the added speed that results from not using object-based security within the system. This does not mean that object-based security should not be implemented, but that Cheyenne will probably not make use of it.

48.4.2.3 System Management

Several major system management components run on the Stone systems in a Cheyenne:

- System management server

  This is the remote procedure call server that executes system management functions.

- System management command-line interface

  This is exactly the same component that runs on client systems. It implements the command-line interface to system management, diagnostics, and database administration functions. When running on a Stone system, it reads commands from the console terminal and writes results to the same console terminal. The interactive DECwindows interface is not supported through the console terminal, because the console terminal is a character-cell device.

  Providing the command-line interface on the console terminal makes it possible for secure sites to disable remote system management. Note that most sites can be expected to have an extended service processor; such sites can run both system management interfaces from the extended service processor.

- Configuration manager and fault monitor

  See Section 48.4.2.6 for details of these components.

48.4.2.4 Network Management

Cheyenne makes extensive use of the following Mica network components:

- DECnet Phase V

  Mica is highly dependent on DECnet Phase V. Cheyenne requires the following DECnet Phase V features:

  — Multiple-rail support

  — Virtual-circuit failover to alternate paths

- DECnet/NI

  For the first version of Cheyenne, it is likely that all client/server communications will take place over the NI (Ethernet). Mica implements DECnet Phase V, which allows us to take advantage of multiple NIs to provide greater throughput and availability between clients and Cheyenne than is possible under DECnet Phase IV.

- DECnet over the CI

  The CI bus provides considerably higher throughput than the NI. Unfortunately, the DECnet /VAX support of the CI is marginal at best. Although it appears that the CI should be the interconnect of choice between clients and Cheyenne, we will not use it for the first version of Cheyenne. The CI will become a practical interconnect when the DECnet/VAX performance is significantly improved.

- Reliable communications service

  RCS is the layer between DBC and DECnet. It multiplexes multiple DDA sessions into a few DECnet logical links. It also identifies client processes to Quartz software for authentication Purposes.

- Remote procedure call support

  Remote procedure calls are used between the system management interfaces and the system management server. The use of RPCs provides a layer of insulation between client systems and Cheyenne system management. This layer will yield us considerable flexibility when we interface new client operating systems to Cheyenne.

- Interprocess communications (both within a Stone system and between Stone systems)

  Cheyenne provides an interprocess communications mechanism for use both within, and between, Stone systems. This mechanism is based on half-duplex message queues. There are two types of message queues: sources and sinks. A source message queue is connected to exactly one sink message queue. All messages sent to the source message queue may be received from the sink message queue. A sink message queue may have one or more source message queues associated with it.

  Local message queues are designed to allow messages to be passed between processes without copying them. This is accomplished by passing the messages in shared memory segments. Remote message queues use SCA over the CI to pass messages. The interface is carefully designed to use the SCA block-data services to transport messages.

- Distributed name service

  Because Cheyenne implements DECnet Phase V, it requires access to a distributed name server. The name server is used to identify other systems in the network.

- Distributed file service

  Cheyenne runs the client side of the distributed file service. The extended service processor runs the server side of the distributed file service. This allows Cheyenne to access the console load device on the extended service processor.

48.4.2.5 Transaction Management

It is clear that Cheyenne must participate in distributed transactions. Such transactions may only involve two nodes, the client and the Cheyenne server, or they may be considerably more complex. An example of a complex distributed transaction is a retail end-of-day roll-up operation. This might involve reading sales records from a set of cash registers, updating centralized financial records, and updating stock records kept at the same site as the cash registers. This is a nontrivial application.

To participate in distributed transactions, Cheyenne must implement a distributed two-phase commit protocol. Because the vast majority of distributed transactions are quite simple, it is necessary that Cheyenne be able to participate in a distributed transaction as the commit coordinator. Because we must support distributed transactions involving more than one Cheyenne, Cheyenne must also be able to participate as a leaf node.

Mica provides Cheyenne with sophisticated transaction management and logging services. These services are initially only used by Quartz. We expect that they will be used by other facilities in future versions of the product (see Section 48.6.2).

Mica provides both local transaction services, and the ability to interact with remote transaction managers. The local transaction services are procedure-based. Mica communicates with remote transaction managers through the yet-to-be-defined corporate distributed transaction management protocol. This protocol defines the interactions required and allowed between distributed transaction managers.

48.4.2.6 Cheyenne Diagnosis and Maintenance

Cheyenne diagnosis and maintenance functions are performed mostly while the system is still running. Hardware maintenance of a highly available Cheyenne may require taking down one of the Stone systems in the server, but the server remains available through the remaining Stone systems.

Diagnosis is performed with one or more of the following tools:

- Automatic diagnostic tools

  These tools always run while the Cheyenne server is up. They constantly monitor faults, and record and attempt recovery when faults do occur.

  — System error and event log

    This log is maintained by Mica on a disk. It contains the following types of information:

    - System configuration
    - Significant media events (for example, disk mounts and dismounts, tape volume mounts and dismounts)
    - A record of every fault experienced by the system (for example, machine checks, software bugchecks, disk errors)
    - A record of configuration changes (for example, shadow-set counterpart replacements, controller failovers)

  — Automatic configuration management running on Mica

    This set of software consists of a fault monitor and a configuration manager. The fault monitor watches for hardware faults recorded in the error log. It applies heuristics to error patterns to attempt to predict hardware failures before they occur. The fault monitor passes such predictions onto the configuration manager.

    The configuration manager responds to external events such as operator-requested configuration changes and predicted device failures. It attempts to adjust the hardware configuration to minimize the effects of such failures. Finally, the configuration manager notifies other software (for example, Quartz and the extended service processor) of configuration changes and nonrecoverable faults.

— Automatic symptom directed diagnosis (SDD) running on the optional extended service processor

The extended service processor is an option that is present if the site has purchased service from DIGITAL. CSSE will supply various SDD tools that will analyze fault patterns to isolate failing field-replaceable units. These tools may also be able to predict field-replaceable units that will soon fail. The CSSE SDD tools are considerably more sophisticated than the fault monitor provided by Mica. The extended service processor will also send failure predictions to the Mica configuration manager. This gives Mica the opportunity to alter the configuration to minimize the impact of the anticipated hardware failure.

— Execution of self-test diagnostics

Each Stone component includes an automatic self-test. This self-test is run when the system is powered up. Mica can also request execution of some of the self-tests. Components that fail self-test are not configured into the system when it boots. The self-test failure is recorded in the system error and event log when the system configuration is recorded.

— Execution of background diagnostics

Idle processor time is soaked up by nondestructive diagnostics that run at low priority.

• Manual diagnostic tools

Most problems are expected to be found by the automatic diagnostic tools. Manual tools are provided for those problems that are not caught by the automatic tools.

Most of these tools run under the PRISM Diagnostic Monitor (PDM), which is supplied with Mica. The PDM, in turn, executes under the system management user interface. Thus these tools can be run through the remote system management interface, or locally through the console. PDM can be used from the console in both the on-line and off-line environments.

— Manual analysis of the system error and event log

An error log-formatting program is provided to view the error log. The program can format and display any error-log record. It can also apply elementary selection criteria to the error log (for example, choose records based on time, device, device class, and record type).

— Execution of on-line and off-line diagnostics

On-line diagnostics are mostly used to exercise and functionally test I/O controllers and peripherals. These diagnostics execute under the PRISM Diagnostic Monitor. All on-line diagnostics allow Cheyenne to continue running. Some require that the device under test be dedicated to the diagnostic. Others allow the device under test to continue to be used by Cheyenne.

In most cases, on-line and off-line diagnostics are the same. The need may arise for off-line diagnostics that cannot be run with other activity on the system. These diagnostics are run on the off-line Mica system, a stripped-down Mica that supplies only enough functionality to run off-line diagnostics and perform elementary system management functions (for example, off-line backup and disk verify).

— Execution of standalone diagnostics

Standalone diagnostics are diagnostics used when the hardware cannot boot Mica. These diagnostics are hardware-dependent and are self-sufficient. Standalone diagnostic execution is controlled from the console.

Software maintenance involves updating the Mica and Quartz software. Software updates are applied to one Stone system at a time, and require rebooting the Stone system to take effect. Each Stone system in the Cheyenne system is updated, one after the other. Thus the Cheyenne server remains available while its software is being updated.

An open question remains regarding the tools that will be provided for altering Cheyenne software. One extreme is to provide no tools, and to require that crash dumps be sent to Engineering for analysis. The other extreme is to provide a sophisticated crash dump analyzer, symbolic debuggers, internals documentation, and training to software specialists, or even customers.

Regardless of what is shipped with the product, Mica and Quartz development require the crash dump analyzer and debuggers. Three debuggers are being built:

- Delta

  This is an elementary high-IPL kernel debugger. It is not symbolic; its interface is through the console.

- Pdebug

  This is an interim remote symbolic debugger based on VAXELN's EDEBUG. It does not allow high-IPL debugging, although it does allow both user-mode and IPL 0 kernel debugging. Its interface runs on a client system and uses DECnet to communicate with the server.

- SDT debugger

  This debugger is being developed for use with the Glacier compute server. It is a sophisticated, easy-to-use, remote, user-mode debugger. It may also allow IPL 0 kernel-mode debugging. The user interface runs on a client system and uses DECnet to communicate with the server.

48.5 Special Challenges

Cheyenne poses several significant technical challenges that must be solved if it is to be successful. These include achieving high availability, support, testing, and meeting internationalization requirements. The following sections discuss these challenges in more depth.

48.5.1 Achieving High Availability

One of Cheyenne's most significant features is that it provides *highly available* access to its databases. High availability refers to the ability for an application to start a transaction at any time. Cheyenne is also *highly reliable*. High reliability assures applications that once they start a transaction, there is a high probability that the transaction will be able to run to completion.

The availability goal for Cheyenne is 100%. CSSE defines an unavailable database as one that an application must wait at least two minutes before it can start a transaction that accesses a database. The goal is for applications to *never* have to wait more than two minutes to access a database. Because hardware occasionally fails, as does software, Cheyenne uses the following techniques to ensure that databases remain highly reliable:

- Replicated hardware

  Much of Cheyenne's hardware is replicated, greatly increasing system reliability. Cheyenne can usually fail over to backup hardware without affecting any transaction. Examples of replicated hardware include disk shadowing, dual-porting disks to multiple controllers, and the presence of multiple processors.

  Some failures of replicated hardware may result in reduced performance, but do not make the database unavailable. An example would be the noncatastrophic failure of a processor. The processor is removed from the configuration and processing continues. The transaction on whose behalf the processor was working might be aborted, but other transactions would continue.

- Multiple Stone systems in a Cheyenne configuration

  The ultimate in hardware replication is achieved by using multiple Stone systems to form a highly available Cheyenne database server. When one Stone system crashes, the others pick up its workload and continue processing. Transactions that were being processed on the failed Stone system are aborted. Note, however, that they can be immediately restarted on the remaining Stone systems.

The Stone systems in a Cheyenne are interconnected using the CI. From Mica's perspective, the Stone systems are independent of one another. Only the interprocess communication, transaction management, Quartz, DECnet, and possibly system management components need be aware that the Stone systems are all part of the same Cheyenne system.

Mica's interprocess communication mechanism operates directly over the CI. This is the mechanism by which software on one Stone system communicates and coordinates with software on the other Stone systems in the Cheyenne. Although the interprocess communication software is able to communicate with the other Stone systems, it is otherwise unaware and unconcerned that the Stone systems are cooperating to form Cheyenne.

The transaction management software is one of the primary components that tie the Stone systems into one database server. Two-phase commit protocols are used extensively to coordinate updates made on behalf of a transaction by more than one Stone system.

Quartz software is what makes the Cheyenne highly available. Although Mica provides various tools that Quartz uses, it is Quartz' responsibility to recover from the failure of a Stone system.

DECnet is used to communicate between client systems and the Stone systems in Cheyenne. DECnet and RCS collaborate to make the Stone systems appear to DBC as one database server.

There may be some system management functions that need to be performed on all the Stone systems in a Cheyenne, rather than just on one Stone system. The system management user interface fans out such requests to the appropriate Stone systems. The system management server is unaware and unconcerned that the Stone systems are cooperating to form Cheyenne.

- Fast restart

  Although we are building the most reliable hardware and software we can, there will be times when an entire Cheyenne system crashes. This may be due to power failure, or may be due to a software error. Regardless of the cause of the failure, the Stone systems in a Cheyenne must restart and be ready to accept new requests in under two minutes.

  This fast restart places significant constraints on how Mica software is designed. For example, it is not practical to mount disks serially; instead, Mica mounts disks using as much parallelism as the disk controller hardware will allow. Decisions like this are widespread in the system. Each must be handled in a case-by-case manner, but parallel algorithms are usually used to meet the two-minute restart requirement.

48.5.2 Support

Cheyenne's primary target market is OLTP applications. Most OLTP applications are regarded as commercial applications. Up to now, most commercial applications have been built on top of other vendors' offerings, with IBM being the primary vendor in the commercial data processing marketplace. DIGITAL has not had a strong presence in commercial data processing. Instead, DIGITAL's strength has been in scientific applications.

The expectations of commercial customers for support differ widely from those of most scientific customers. IBM historically has provided a very high level of support to its large commercial accounts. This support has included on-site software and hardware support personnel; problem-free product installation; guaranteed fast problem response; products designed to gracefully recover from faults and problems; dedicated sales staffs familiar with the customer's industry; and so on. DIGITAL, on the other hand, historically has provided off-site support personnel; fault-intolerant products; and smaller, less-specialized sales staffs. OLTP customers will expect levels of support from DIGITAL similar to that they have come to expect from IBM. This poses special challenges for our engineering, manufacturing, support, and sales organizations.

Engineering must design Cheyenne to be easy to install, and build in the capability for recovering from most faults transparently to the application. Cheyenne should be as easy to use as possible, because this will reduce the product's apparent complexity and thus the customers' support needs. Support needs are further reduced because Cheyenne is DDA-compliant, allowing Cheyenne to be used with

existing applications and tools, and allowing customers to take advantage of their familiarity with existing DDA products.

There will need to be a way to quickly fix bugs for customers. Most DIGITAL products rely on workarounds and FCOs to fix hardware problems, and workarounds and updates to fix software problems. Because Cheyenne is a closed system, there is a good chance that workarounds will be less applicable as short-term fixes than in existing products. Hardware changes have inherently long lead times, so Cheyenne will require a fast way to transport emergency software updates to customer systems.

The sales and support organizations are faced with a challenge: the need to evolve new methods for selling and servicing Cheyenne systems. They will be competing head-to-head with vendors that are already familiar to the customers, whereas DIGITAL will be the new guy on the block. DIGITAL's OLTP offerings will have to be perceived as clearly superior to other vendor's products if they are to gain market share.

48.5.3 Testing

Cheyenne's reliability and availability goals exceed those of any other major DIGITAL product. Meeting those goals will be difficult; demonstrating that we have met them will be even harder. The following strategies are used to meet these goals:

- Development of a comprehensive test plan for each Mica and Quartz component

  These plans cover functional testing (including stress tests) and regression testing. The developers working on the components are responsible for implementing the tests. Once built, the tests become part of the permanent Cheyenne test suite.

- Development of a comprehensive test plan for single and multi-Stone Cheyenne configurations

  This plan includes functional testing, stress testing, and fault-insertion testing. Special emphasis on fault-insertion testing will help ensure that the multi-Stone failover capability works.

- Development of a system performance testing strategy

  This serves two purposes: providing exercisers to stress test the system, and providing much-needed benchmark data for the sales force.

Testing is such an important facet of the overall Cheyenne program that all developers will participate. Additionally, the Mica and Quartz development groups each have subgroups dedicated to the testing function.

48.5.4 Ease of Use and Internationalization Requirements

Cheyenne always acts as an agent of a client. For example, Cheyenne will not spontaneously start a transaction to roll-up the year end financial results. Instead, the year-end roll-up application runs on a client system and runs requests on Cheyenne to query and update the financial database.

Cheyenne should appear to users as if it is an extension of the client system. This has two benefits:

- The similarity to the already familiar client system reduces training time and reduces mistakes.
- Eliminating the need to think of Cheyenne as a separate shared entity makes it easier for many users to conceptualize Cheyenne. This, in turn, makes it easier to interact with and manage the server.

One of the primary means by which a system appears comfortable to its users is to present information in the user's native language. Another is build the system so that different interfaces can be used by different users. Each interface is designed to be ideal for its intended user. Section 48.4.1.3 described three classes of tools; each of the three classes of tools is used by different users, and follows the interface model most familiar to those users:

- Generic DDA tools

  These tools are supplied as layered products, and thus conform to models appropriate for the various layered products.

- Cheyenne-specific DDA tools

  These tools are mainly used by database administrators. Because database administrators use both of the other classes of tools, these tools follow the guidelines for the other classes.

- Cheyenne database administration and tuning tools

  These tools are used by database administrators and system managers, and all share a common interface style. Two types of interfaces are supported: a command-line interface, and an interactive DECwindows interface. The command-line interface is chosen to resemble that of the client system (initially VMS). The interactive interface is chosen to be easy-to-use and informative. As such, it currently has no existing model in DIGITAL.

Each tool must conform to DIGITAL's internationalization requirements. Engineers must consider both input and output internationalization requirements when designing tool interfaces. The manual *Producing International Products*, available from the International Products Group, contains valuable guidelines and suggestions for creating products that are easy to translate into many languages. In addition, a paper containing specific guidelines for Cheyenne will be available in February to Mica and Quartz developers. This paper describes the techniques to be used by developers to insure that Cheyenne meets the International Products Group's requirements.

48.6 Related Products

48.6.1 Other DIGITAL Products

The following list enumerates many of the products that are related to Cheyenne:

- VMS

  VMS is the primary operating system for VAX computers. It is the only system supported as a directly connected Cheyenne client at FRS.

- DECnet/VAX

  DECnet/VAX is the primary local- and wide-area network for VMS.

- ULTRIX-32

  ULTRIX-32 is another operating system for VAX computers. It is gaining in popularity, especially in the scientific workstation marketplace. Although it is not supported as a Cheyenne client at FRS, we expect that it will soon after be a supported client.

- DECnet/ULTRIX

  DECnet/ULTRIX is a local- and wide-area network for ULTRIX-32. It provides the primary means by which ULTRIX-32 systems communicate with VMS systems.

- Glacier

  Glacier is a compute server based upon Moraine and Mica. Both Glacier and Cheyenne are being developed in parallel and share a great number of components.

- Moraine

  Moraine is the precursor to Stone. Stone uses the same power, packaging, I/O, memory, and console subsystem as Moraine, but Moraine uses scalar/vector processor modules, while Stone uses shadowed scalar/scalar processor modules.

- Rdb/VMS

  Rdb/VMS is a DDA-compliant relational database management system. It is packaged as a layered product on VMS. Rdb/VMS covers the low end of the relational database marketplace for VMS. Cheyenne covers the middle-to-high end.

- Rdb/Star

  Rdb/Star is a product in development that supports true, distributed, DDA-compliant databases. It will allow applications to access databases implemented on Rdb/VMS and Cheyenne.

- VIDA

  VIDA is a DIGITAL layered product for IBM and VMS systems. It allows IBM-resident applications to access DDA databases. VIDA accesses Cheyenne databases through the SNA gateway software running on a VAX.

- ACMS

  ACMS is a transaction processing monitor for VMS systems. It it a DIGITAL layered product. ACMS applications will be able to access Cheyenne databases.

- Intact

  Intact is another transaction processing monitor for VMS systems. It is currently under development as a DIGITAL layered product. Intact applications will also be able to access Cheyenne databases.

- Datatrieve and others

  VAX Datatrieve is a fourth-generation query language processor running on VMS. It, along with similar products, allows users to generate ad-hoc queries against DDA databases.

48.6.2 Future Versions of Cheyenne

The first version of Cheyenne is a DDA-compliant relational database server. No user application programs execute on Cheyenne. Version 2.0 of Cheyenne will continue to act as a database server, but it will also run a transaction processing monitor. User application programs will run under the transaction processing monitor. There will not be any provision for interactive programs (there will not be a command-language interpreter).

The development environment for the first two versions of Cheyenne is VMS. Other development environments may be added (for example, ULTRIX). Neither of the first two versions support general interactive timesharing or batch-mode execution. Cheyenne will not host a development environment until Version 3.0, at the soonest.

Some of the many component improvements expected in the second version of Cheyenne include:

- RMS—Support for additional file organizations (for example, indexed file support), support for recoverable files
- DECnet—Support for use of the CI as a LAN
- RPC—Support for heterogeneous systems
- DFS—Support for DFS Version 2.0
- Quartz—Improved performance and reliability, along with support for additional data types and (possibly) alternative data models

48.7 Issues and TBD

1. Client/server communications

   The model for client/server communications is not clear. The distribution of functions between the database communications and RCS layers has yet to be done. In addition, the corporation's recent efforts in support of the Distributed Transaction Architecture (DDTA) make it possible that the model for client/server communication will be radically different that what is proposed in this chapter.

   This issue should be resolved by 15 March 1988.

CHAPTER 49

TRANSACTION SERVICES

49.1 Overview

This paper summarizes the transaction services for the Mica operating system, which provide the following functionality:

- Common logging support with log checkpoints
- Basic transaction control services (*exec$start_transaction*, *exec$commit_transaction*, *exec$abort_transaction*) with Two-Phase Commit
- Recovery from the common log after system failure

Note that in the processing of commit, abort, log checkpoints, and recovery, the Mica services are only providing the central control; facility-specific processing of these events is coordinated by Mica, but always performed by the facility itself.

For the FRS products, this support would be shipped with the Cheyenne database server only. However, it will not be bound to the database server in such a way as to preclude its inclusion in a subsequent release of the compute server or other Mica-based systems.

\This overview does not attempt to define or explain any of the fundamental concepts of transaction processing. Please see the references at the end of this section. The chapter will ultimately attempt to be a little more self-standing.\

49.1.1 Goals

The goals of the transaction services for Mica are as follows:

- Provide a framework which, together with the Quartz software, supports the recoverability and performance requirements of the Cheyenne database server, consistent with the definition of the goals and requirements of the database server product. (For a summary of these goals and requirements, see the Cheyenne chapter overview.)
- Support the recoverability requirements of other Mica facilities which will be present at both FRS and beyond.
- Design these services in such a way as to allow Mica to ultimately become the basis for a PRISM-based TP platform.
- Attempt to define the structure of the transaction services in such a way as to not dictate the policy or algorithms of the individual recoverable facilities, nor to subsume any facility-specific processing.

49.1.2 Functional Overview

Transactions are represented in Mica by transaction objects. The object architecture provides efficient mechanisms for naming, identifying, and protecting transaction objects. The object ID is used for fast access to a transaction object within a single system. For distributed transactions, a global transaction ID is used as the name of the transaction object, which is formed from the coordinator node identification, and a serial number for that node.

The standard ACL-based protection mechanism of the object architecture may be used to protect who can control or write log records on behalf of a transaction; or else a dedicated system like Cheyenne may eliminate protection checking by simply not using ACLs.

The object architecture also provides a very efficient and protected interface to the transaction control services: effectively, a synchronous call to a procedure in kernel mode. The use of a synchronous interface dictates an implementation utilizing a large number of threads, each servicing only one transaction at a time. Threads and transaction objects do not have to be created and deleted for each transaction. A large transaction system may use hundreds of threads, each creating its own transaction object during initialization, and reusing this transaction object to service successive transactions. These threads may all being driven off of shared transaction request queues.

This thread-based design is consistent with the philosophy of Mica. Note the following benefits over a solution where threads service multiple transactions in parallel via ASTs:

- Maximal parallel processing is utilized in a large multiprocessor system with a thread-based solution. An AST-driven solution introduces unnecessary serialization when a number of ASTs are queued to a single thread.
- Given the knowledge of when threads commit transactions, the kernel can distinguish short-running transactions from long-running ones, and optimize scheduling in transaction systems, by decaying the priority of long-running transactions, and restoring the priority on commit.
- Transient faults handled by Mica, such as nonrecoverable machine checks, may be limited to affecting at most one transaction at a time.
- The coding of transaction programs is greatly simplified, through the use of straightforward sequential programming.

The central support for the Mica transaction services is implemented by two components: the transaction object service routines and the recovery manager. Figure 49–1 depicts these two components, and how they fit into Mica with transaction programs and recoverable facilities.

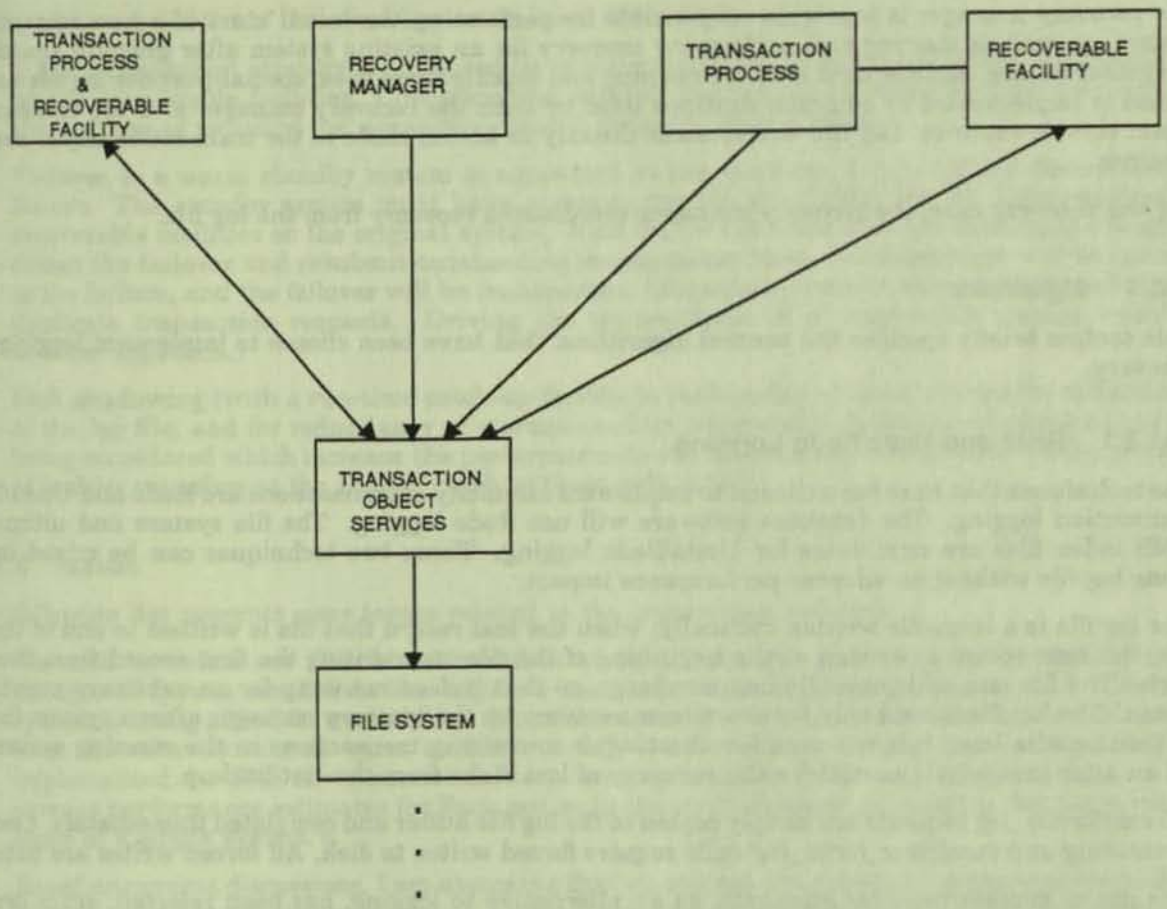
The Mica transaction services have two types of clients: transaction processes and recoverable facilities. In some cases, such as the Quartz processes in the database software, these two processes may be one in the same. Both transaction processes and recoverable facilities call the transaction services. In addition, recoverable facilities receive asynchronous "calls" from the transaction services via message queues. \The exact queueing mechanism is TBD.\

The interface between transaction processes and recoverable facilities is not dictated by the transaction services, but rather, determined by the recoverable facility. Recoverable facilities, when separate from the transaction process, will typically be protected subsystems.

The file system in Figure 49–1 is the normal Mica ODS-2+ file system. (The file system may also interfaces to the transaction services as a recoverable facility, but this is not shown.)

The transaction services and recovery manager are capable of supporting multiple recovery systems at once. A recovery system is a set of recoverable resources serviced by a set of recoverable facilities and a log file. Thus recoverable facilities must also expect to be called for multiple recovery systems. For performance reasons, there is normally only one recovery system per Mica system, but multiple recovery systems may run on a single Mica system after failover of one of the recovery systems from another Mica system.

Figure 49–1: Transaction Services Block Diagram



The following sections present a brief discussion of the transaction object service routines and the recovery manager.

49.1.2.1 Transaction Object Service Routines

The transaction object service routines provide a set of procedures called by both transaction threads and recoverable facility threads. In addition, the transaction object services notify recoverable facilities of asynchronous events via predeclared message queues.

Transaction threads call the transaction services for *exec$start_transaction*, *exec$commit_transaction*, and *exec$abort_transaction*. Recoverable facilities call the transaction services to write log file records, declare message queues, and signify completion of operations requested via the message queues.

The message queues are used to effectively "call" recoverable facilities with events related to transactions, checkpoints, recovery, and maintenance operations.

49.1.2.2 Recovery Manager

The recovery manager is a process responsible for performing the initial start of a new recoverable system, as well as starting and performing recovery for an existing system after graceful shutdown or system failure. RMS is used only for opening and closing these files; special-purpose log file record access is implemented by common routines used by both the recovery manager and the transaction object service routines. Log file writes occur directly in kernel mode in the transaction object service routines.

For the recovery case, the recovery manager coordinates recovery from the log file.

49.1.3 Algorithms

This section briefly specifies the central algorithms that have been chosen to implement logging and recovery.

49.1.3.1 Redo and Undo/Redo Logging

The techniques that have been chosen to implement atomicity of transactions are Redo and Undo/Redo transaction logging. The database software will use Redo logging. The file system and ultimately RMS index files are candidates for Undo/Redo logging. These two techniques can be mixed in the same log file without an adverse performance impact.

The log file is a large file written cyclically; when the last record that fits is written to end of the log file, the next record is written at the beginning of the file, overwriting the first record from the last cycle. The file size will generally be quite large, so that it does not wrap for an arbitrary number of hours. The log file is used only for short-term recovery by the recovery manager after a system failure without media loss. It is not used for aborting or commiting transactions in the running system, or as an after image (AI) journal for the recovery of lost disks from the last backup.

All *exec$write_log* requests are simply copied to the log file buffer and completed immediately. Commit processing and *exec$force_write_log* calls require forced writes to disk. All forced writes are batched.

The use of process pairs for atomicity, as an alternative to logging, has been rejected, since process pairs entail a much greater run-time overhead for a small improvement in recovery time. It is also nearly impossible to build an efficient solution based on process pairs that makes recoverability transparent to the transaction program writer.

49.1.3.2 Two-Phase Commit with Presumed Abort

The commit protocol which has been chosen is standard Two-Phase Commit protocol (2PC) with the presumed abort (PA) optimization. This protocol dictates the interaction between the transaction object service routines and all recoverable facilities within a single node during commit processing. It also dictates the message protocol for commits in the distributed TP case where a transaction modifies recoverable resources on multiple nodes in a network.

The presumed commit optimization to 2PC was rejected since it requires an additional forced write on the coordinator.

49.1.3.3 Other Techniques

Following is a brief list of the other key techniques that will be employed:

- A fuzzy checkpoint strategy is employed to periodically force the flushing of cache data managed by the recoverable facilities. Checkpointing expends a little bit of extra run-time overhead for the purpose of reducing the worst-case time to recovery from a failure.
- Failover to a warm standby system is supported as the quickest way to recover from a system failure. The standby system must have access to the log file disk(s) and all disks required by recoverable facilities on the original system. Note that if the front ends are intelligent enough to detect the failover and resubmit outstanding transactions, then no transactions will be lost due to the failure, and the failover will be transparent. (Sequence numbers may be used to eliminate duplicate transaction requests. Driving the transactions off of recoverable queues would be another approach.)
- Disk shadowing (with a run-time catch-up facility to replace failed disks) is used for redundancy of the log file, and for redundancy of storage used by recoverable facilities. \Optimizations are being considered which increase the performance to shadowed disks as well as solve the problem of broken transfers at the time of crash without safe RAM.\

49.1.4 Issues

The following list presents some issues related to the transaction services:

- We need to define our distributed/multibox model. I propose that if a single, possibly fully-configured system can deliver the required performance for a given application, and if another system is being added for availability, then the transaction load should be handled by one system with the other system as warm standby. A full distributed transaction model should only be implemented for FRS if we cannot achieve our performance goals on a single system. \The current performance estimates for Rock put us in the right ballpark for meeting our performance goals in a single system.\
- Based on current discussions, I am assuming that we are not attempting to implement centralized locking. Thus, in a distributed system, or in a single system with multiple facilities implementing their own locking, there is a potential for distributed deadlocks. Given the distributed deadlock detection techniques proposed by Bernstein, et al., this may be acceptable. Alternately, a more centralized locking strategy could be considered for a subsequent release of Mica.

  This should not be a problem for the Cheyenne database system.
- It would be convenient to rely solely on shadowing for redundancy of disk-resident data. If it is necessary to implement recovery of a lost disk by applying an after-image (AI) journal to a recent backup, then this AI journal is best written by the recoverable facility during its commit processing. If it is decided for Cheyenne that the Quartz software must implement AI journaling, then many things must be thought out very carefully, such as coordination of the AI journal and the log file, resynchronization of the AI journal and log after normal recovery without media loss, and recovery of lost media from the AI journal and subsequent resynchronization with the log, and so on.
- A distributed system requires a distributed security model. We do not have one now.
- I assume that the ODS-2+ file system must also be implemented as a recoverable facility. Is this true?

49.1.5 Bibliography

The first reference is the most complete and up-to-date overview I have seen. The second describes two optimization techniques not found in Bernstein. The third is a short easy-to-read overview.

Bernstein, Philip A.; Hadzilacos, Vassos; Goodman, Nathan; *Concurrency Control and Recovery in Database Systems*, Addison-Wesley Publishing Company, Reading, MA, 1987.

Mohan, C.; Lindsay, B.; "Efficient Commit Protocols for the Tree of Processes Model of Distributed Transactions", *Proceedings of the Second ACM Symposium on Principles of Distributed Computing, 1983.*

Gray, Jim, *The Transaction Concept: Virtues and Limitations, Tandem Technical Report 81.3*, Tandem Computers Incorporated, Cupertino, CA, June, 1981.

Compute Server

This set of chapters describes the components of Mica that provide compute server support.

CHAPTER 50

GLACIER OVERVIEW

50.1 Overview

The *Glacier* compute server provides users access to the vectorized, parallel-computing, high-bandwidth I/O environment inherent in the Mica operating system and Moraine hardware. Glacier is explictly designed for applications that can be characterized by long mean time of execution, intense CPU utilization, and/or large data sets. Such applications include circuit simulation, reservoir simulation, finite element analysis, and molecular modeling.

The Glacier system can be viewed as a "compute accelerator" for the client system. The Glacier software design is based on an integrated client/server interface through which applications can be developed and executed. The system appears to the user as simply a higher performance client. This characteristic of Glacier is referred to as *seamlessness*.

As an application executes, the underlying Glacier software (client- and server-based) interacts to provide window support, distributed file services, context services, and management functions, among others. The client communicates with the server via DECnet using an Ethernet or Computer Interconnect (CI) link.

System management, operator communication, and performance monitoring and tuning facilities are activated and controlled through the client system. This allows for flexible compute server management. A remote operator or system manager can support multiple compute servers from a single client system.

50.1.1 Goals

The Glacier product has a set of goals that extend the basic goals of the underlying Mica operating system.

- Provide client/server integration such that server application activation is initiated from the client system using standard client commands. Context established in the client enviroment is communicated to the server during program execution.
- Provide a compute server programming environment that conforms to the Application Integration Architecture (AIA).
- Support multiple operating systems as clients.
- Require no modifications to the native client operating system.

50.1.1.1 Client/Server Integration

The compute server is a logical extension of the client's computing resources. The client and compute server cooperate to provide the feel of a single environment to the user. This is accomplished through the client's user interface, bidirectional shared file access, and by providing server access to client environmental context such as logical names. The compute server may have local file storage for increased I/O bandwidth.

50.1.1.2 Application Integration Architecture

The Application Integration Architecture (AIA) is designed to provide a system-independent representation of system services and high-level functions such as window management through DECwindows. AIA includes basic system services such as math libraries and multiple thread support. Higher-level services, such as RMS, and access to environmental context are also represented through AIA. The application programmer is shielded from the internal nature of Mica system services. The result of AIA conformance is application portability.

50.1.1.3 Multiple Operating Systems Support

The client/server interface is designed for support of multiple, heterogeneous operating systems. This is accomplished by implementing a well-defined, extensible interface that ties the two enviroments together. Those AIA services that require client support use RPC to call native services in the context of the appropriate client. The extent of cross-system RPC compatibility factors into the ability to support heterogeneous systems.

50.1.1.4 Client Modification

All Glacier client software is layered on the client without modifying the client operating system. Modification of the client's native system would create complex dependencies between the client operating system and the compute server. This would require Glacier software to be in lock step with each client system software release or it would necessitate the negotiation of permanent changes to the client operating system. Modification to the client system also makes the support of new client operating systems significantly more difficult.

50.2 Target Customer Base

Glacier's primary target market is the compute-intensive scientific and technical community, currently being served by vendors such as IBM and Cray, with strong inroads being made by Convex and Alliant. Apollo and Sun are now entering this market as well.

Applications in the scientific and technical market cover diverse areas such as modeling, finite element analysis, earth resources, and fluid dynamics. Although these applications span a wide variety of scientific disciplines and program behaviors, some general observations can be made about their computational needs, which include high performance program execution, fast and efficient vector and matrix manipulation, fast I/O, multiuser access, security capabilities, and quality vendor support.

Although market analysis determines the types of applications that will run on Glacier, it does not describe the people who will interact with Glacier. It is useful to consider how various classes of Glacier users will view Glacier, in order to tune the system interfaces for them. The following sections describe the ways in which these various classes of users will interact with Glacier.

50.2.1 Application Users

Today, some users run their scientific and technical applications on Convex, Alliant, IBM, and Cray systems. Other users, at less affluent companies, run applications on less-capable systems, which greatly increases their execution time. Still other users have access to supercomputing systems through complicated gateway programs or procedures.

Application users are interested primarily in results. They expect a simple interface, consistent with the rest of the often minimal command set they use, to request program execution.

When a Glacier system is installed, application users will notice only that their application programs run significantly faster than previously. Job submission and execution will be seamless.

50.2.2 Application Developers

Application developers need to have some level of understanding of the Glacier system. Specifically, they need to know the following:

- Significant system features that can be utilized to the advantage of the application

  Developers may want to restructure the code to take advantage of vectorization and program decomposition.

- Commands to compile and link programs targeted for Glacier

  The Glacier client user will have access to commands which are compatible with the client operating system. For VAX/VMS, Glacier program development tools will be accessed with approved DCL commands. Developers using ULTRIX client systems will access commands compatible with the ULTRIX environment.

50.2.3 System Managers

A system manager is responsible for the hardware, software, and data integrity of the Glacier system, and for administrative duties, such as maintenance of the user authorization database. To perform these duties, the system manager must be aware of Glacier as an entity distinct from the client system.

However, the interfaces by which Glacier is controlled are tightly integrated with the client environment, such that Glacier appears to the system manager to be an extension of the client system.

50.2.4 Operations Staff

Operators are responsible for the day-to-day operation of Glacier, with duties defined by the system manager. Typically, operators perform backups and their storage, and may be the first level of interaction with failing hardware components.

Operators interact with the Glacier system through a client interface, and, like the system manager, view Glacier as an extension of the client system.

50.2.5 Software Support Personnel

Software support personnel are the first DIGITAL personnel that a customer contacts for support and technical information regarding Glacier. The software support specialist will typically be called on in two instances: for questions regarding program vectorization/decomposition and for problem resolution in the event of system failures.

Providing quality customer support for vectorization and decomposition requires knowledge of the concepts and mechanisms utilized within Mica and the language run-time libraries which implement the underlying support for user programs. The specialist needs to be skilled in utilizing application analysis tools such as PCA (Performance and Coverage Analyzer) as aids in resolving applications problems.

In order to support difficult system problems, the specialist must understand the operating system internals, including system design and synchronization mechanisms. It is expected, however, that specialists will not provide on-site corrections to Glacier or Mica software; rather these fixes will be generated by engineering once the problem has been identified and a solution developed. The modular design of the Mica operating system and its extensive working design document will substantially reduce the job of problem isolation.

50.2.6 Hardware Service Personnel

Hardware service personnel install and repair systems. They need to know how to install the hardware configuration and the system software. In addition, hardware service personnel are responsible for running the USE (User-level System Exerciser) to ensure correct system installation, and for executing diagnostics to identify failing components when failures occur.

50.2.7 Internal Software Developers

DIGITAL internal software developers build software products for Glacier, such as language compilers and run-time systems, and application analysis tools. Developers require complete functional interface specifications to which they can build their software. Run-time debug and analysis tools aid in debugging and verifying the correctness of the product execution.

Most software developers building products will utilize the Application Integration Architecture (AIA) interfaces to the system to promote software portability across PRISM and VAX systems. Only a small number of products, such as the debugger and performance analysis tools, require system interfaces not provided by AIA, or utilize direct interfaces to the operating system.

50.3 First Revenue Ship Applications

Although Glacier satisfies many of the computational needs of its target markets, it is imperative that a suite of applications used within these target markets be available at Glacier FRS.

To address this need, a program is being put in place to insure that key applications from several application segments are ported to the Glacier computing platform. Activities within central engineering, the product marketing groups, and the field organization are critical to the success of this program.

50.4 Glacier Components

The following sections describe the various components of Glacier, which include: client hardware, client software, server hardware, and server software.

50.4.1 Client Hardware Components

The primary interconnect used by client system hardware to connect to Glacier systems is the Ethernet (NI). DECnet and TCP/IP networking protocols utilized over the Ethernet provide a high-speed interface to Glacier. For the FRS Glacier product, Ethernet and DECnet provide the only supported network interfaces.

The CI will become the interface of choice when connecting large client systems (for example, VAX 8800) to Glacier and may be used for higher throughput for client communications than the NI can provide. Glacier support for the CI will be provided in a subsequent release.

Other than the network interconnect, Glacier places no hardware requirements on the client system. Other client system hardware components which may be utilized with Glacier include:

- Bitmap displays

  Bitmap displays on client systems supporting DECwindows may be used by Glacier applications for high resolution display. To the end user, the application appears to be executing locally on the client system.

- Character-cell terminals

  Glacier applications may be started from character-cell terminals connected to client systems. Application terminal output is directed to the terminal through the Glacier client system software, providing seamless application behavior for the user.

- Mass storage

Glacier client systems need not contain large amounts of mass storage. The only requirement for client system mass storage is storage for the Glacier client system software. Through the use of distributed file services, virtually all other mass storage requirements can be satisfied by use of storage on Glacier.

50.4.2 Client Software Components

Glacier requires a compute server client system to support several software components. These components are layered on top of the client's native environment. Several goals have been factored into the design and use of these components:

- The Glacier components must not require changes to be made to the underlying client operating system or to the software bundled with that operating system.
- The environment presented to the application user while using the compute server must appear as nearly as possible like the native client system environment. Relevant context established within the client system must be conveyed to the compute server during program execution. These goals are the initial basis for seamless computing.
- The software interfaces between the client system and the compute server are based on either corporate or industry standards. For example, DNA, RPC and DFS. This allows future client systems to be added with relative ease. Failing standards, the software interfaces used to integrate the systems are closed, internal mechanisms which could be replaced by standard components in the future.

50.4.2.1 Software Run-Time Environment

The application user's run-time environment is at the center of the design for seamless computing. Several software components aid in this integration. The cornerstone component is the *client context server* (CCS). Its name indicates the primary function of the component: to deliver the context previously established in the client system environment to the running application. For example, logical name translation is done within the client context at the time the application running on the compute server references a logical name.

A substantial list of context services are provided by the CCS. The goal of these services is to allow the invocation and execution of the application program to proceed as if it were running on the client system. Below is a partial list of services:

- Application activation
- Command line parsing
- Logical name creation and translation
- Standard input and output interaction for support of character-cell terminals, command procedures, and batch file submission
- Application suspension and resumption via client interrupt (for example, CTRL/C)
- Debugger activation (typically, this service would not be used by the application user, but is required by the application developer)
- Client defaults such as current directory path and terminal type
- Status value manipulation including exit status emission

A second critical component is Glacier's windowing software. This functionality is provided by DECwindows using the industry-supported X protocol. DECwindows provides integrated, workstation-based, client support, as well as application portability.

To round out the seamless environment, disk-based mass storage devices are reciprocally accessible by the client or compute server system. That is, disks connected to the client system can be accessed as if they were connected to the compute server. Likewise, disks directly connected to the server may be accessed locally from the client system. The software providing this service is considered an underlying mechanism. See Section 50.4.2.4.3 for further details.

50.4.2.2 Software Development Environment

The Glacier product provides a number of software development tools used to produce application programs. These tools include a host of programming languages, a linker, a librarian, and a debugger, among others. In general, these tools run directly on the Glacier system and are accessed by client-based command interfaces. Using the CCS services allows software development to take place using command syntax that is familiar to the software developer. While it is not an FRS goal to provide seamless compute server interaction for software developers, the process used to develop applications for the Glacier environment is very similar to that of the native client system. Example 50–1 shows the compilation, linking and execution of a typical Glacier FORTRAN program in the VAX/VMS environment. More details on the Glacier software development tools may be found in Section 50.4.4.2.

Example 50–1: Typical Glacier Program Development

```
$ FORTRAN/GLACIER APPLIC$SOURCE:PROGRAM
$ LINK/GLACIER/EXECUTABLE=APPLIC$IMAGE:PROGRAM APPLIC$SOURCE:PROGRAM
$ RUN APPLIC$IMAGE:PROGRAM
```

Note that the commands in Example 50–1 are for explanatory purposes only. The command qualifier names are to be determined.

As with the application user's environment, the application developer can use Glacier's bidirectional disk services to access source files or target output files.

50.4.2.3 System Management

All Glacier system administration is difected from a consistent system management user interface (SMUI). In the FRS product, this interface is command line oriented. The interface may be accessed from either an authorized client system or the Glacier system console. In future versions, the system management user interface will support a window interface.

The Glacier SMUI incorporates many of the latest concepts in both local and remote system administration. The interface presents a task-oriented approach to such problems as authorizing new system users and configuring network topology. While this interface is not seamless with the client's system management interface, it is both culturally compatible with VAX/VMS functions and follows the evolutionary path planned for VMS.

Below is a partial list of functions accessible through SMUI:

- User authorization and authentication
- Configuration management
- Network management
- Performance monitoring
- Operator communication
- Diagnostics
- System backup and restoration

50.4.2.4 Underlying Software Mechanisms

The Glacier system relies on several underlying software components that must be provided by the client system for the layered Glacier software to operate correctly. These components are related primarily to network communication.

50.4.2.4.1 Network Support

For a system to act as a compute server client, it must support either DECnet, using the NSP transport protocol, or TCP/IP. For FRS, only DECnet-based clients are supported. Likewise, only Ethernet (NI) connections are provided. Future releases may provide support for TCP/IP on the NI connection and DECnet on computer interconnect (CI).

50.4.2.4.2 Remote Procedure Calls (RPC)

Remote procedure calls are the mechanism used to communicate between client and server systems. For the FRS product, remote procedure calls are used as an underlying mechanism and are not available for general application use. Once the corporate RPC architecture is in place, future releases of Glacier will provide a general RPC facility.

The Mica RPC facility is designed to interoperate with the VAX RPC protocol. The VAX RPC facility is used by several Glacier components in support of VAX/VMS client systems. These components include the client context server, the system management user interface, and the performance monitor.

To achieve the goal of supporting a heterogeneous set of client systems, a common RPC mechanism must be available. Currently, VAX RPC is only available for VAX/VMS, although a corporate RPC architecture is being developed. A plan for migrating Mica RPC from the VAX RPC protocol to the corporate RPC protocol will be executed to achieve corporate RPC interoperability.

50.4.2.4.3 Served Disks

Fundamental to Glacier's seamless environment is the ability to access client disk devices from the server and to access server disk devices from the client system. This presents an integrated view of system resources to the user. A user accesses the files located on a remote or "served" disk, as if the device were connected to the local system.

Two such facilities are in widespread use: Distributed File Services (DFS) for VAX/VMS systems and Networked File System (NFS) for ULTRIX systems. Each of these facilities supports their native file systems. For FRS, a DFS implementation for Glacier supports VMS client systems. Future versions *may use* NFS to provide support of ULTRIX and general UNIX client systems.

50.4.3 Server Hardware Components

Glacier hardware platforms have several characteristics that address the requirements of the target customer base:

- Hardware vectors

  Hardware vectors provide the capability to operate on up to 64 elements simultaneously. Vectors are utilized by compilers to significantly enhance the performance of programs operating on large matrices, which are used by virtually all of the target applications.

- Symmetric multiprocessing

  Through symmetric multiprocessing, multiple processors provide enhanced capabilities in two dimensions:

  — In conjunction with program decomposition by compilers, utilizing multiple processors concurrently to run a single program significantly reduces the elapsed time required to complete the program execution.

— Multiple computational elements permit the system to service more users concurrently, extending the overall capacity of the system.

- High throughput I/O subsystem

  The Glacier I/O subsystem is based on the XMI bus. Up to two XMI buses are supported in a single Glacier package.

- Pipelined execution and a large number of registers

  The PRISM architecture supports pipelined instruction execution on a single processor. A very large set of general purpose registers available to compilers, in conjunction with compiler knowledge of the hardware run-time platform, permit compiler code generation techniques that fully utilize the hardware for optimal program execution.

50.4.3.1 FRS Hardware Configuration

The first revenue ship configuration is built on the Moraine platform, which is a CMOS-2 implementation of the PRISM architecture. Moraine has:

- Two XMI buses for increased I/O capacity. The I/O system is capable of delivering 100-Mbytes/sec throughput.

- Up to four scalar/vector processor pairs each providing 15 VUPs scalar performance and an estimated performance on the double-precision 100x100 Linpack benchmark of 12 Mflops.

  Each scalar/vector processor has a 128-Kbyte scalar cache and a 2-Mbyte vector cache. Most problems that run on Glacier should fit in cache. Larger problems which do not fit will take advantage of the 80-Mbyte/sec/processor memory throughput. The scalar and vector caches use a write-back-to-memory strategy for improved performance.

- A nine-way, fully-connected CMOS-2 crossbar switch for connections between processor, memory modules, and an I/O port.

  Crossbar connection of processors and memory modules avoids the contention of multiple processors connected to a single system bus, providing increased per-processor bandwidth.

- An *n+1* redundant power supply and integral motor generator set.

  These components greatly improve system reliability, serviceability, and availability. The CEAG power supplies provide hot swap and optional *n+1* redundancy. The motor generator set provides power conditioning and increased efficiency, improving availability and reducing the required power supply and overall system size.

  (Although this high availability is not a requirement of the Glacier target markets, it is being implemented in Glacier to provide a platform shakeout for Cheyenne, a database system built on Stone, the CMOS-3 Moraine follow-on platform. Customers utilizing the Cheyenne database system will be putting the well-being of their entire businesses in the hands of the Cheyenne system. As such, these customers want field-proven hardware and software. Shaking down Mica and the hardware platform with Glacier provides the in-field experience Cheyenne customers require.)

The initial release of Glacier uses DSA-1 disks and tapes for mass storage. All DSA-1 disks are supported, although we exepect that only RA70 and RA90 disks will actually be used. The TA90 tape drive is the only drive capable of providing the high performance required for backup of large disk subsystems. DSA-1 devices are attached to Glacier systems through the HSX (Wildcat) controller.

Client systems communicate with Glacier via Ethernet (NI) or computer interconnect (CI) interfaces, with CI support being delivered in a post-FRS version. Additional NI interfaces may be added to Glacier for expanded throughput.

50.4.3.2 Follow-On Configuration

Stone, the follow-on platform to Moraine, will deliver approximately 20 to 25% additional computational capability through the use of CMOS-3 technology in the Moraine package.

50.4.4 Server Software Components

The Glacier server software components are designed with two basic goals:

- To ensure server integration into a client environment such that the server appears to be a seamless extension of the client system
- To ensure that applications written using the Application Integration Architecture are extremely portable to and from other systems supporting the architecture

50.4.4.1 Software Run-Time Environment

The run-time environment is key to achieving the goals of the Glacier system. This layer of software provides the software functions that integrate both the Mica services and the remote client services. Within this layer, services such as file/record access, system services, and windowing support are found. All of the run-time software components are available to the software developer.

The run-time software available to applications is designed around a software architecture common to PRISM/Mica and PRISM/ULTRIX. This common software architecture:

- Permits applications built for one target system platform to be readily ported to another
- Provides the capability for supporting a larger number of third-party applications on both software platforms, satisfying a broader spectrum of customers
- Permits DIGITAL to build a single set of layered software products which execute on either software platform with a much smaller engineering investment

50.4.4.1.1 Application Integration Architecture

The interface to Glacier's run-time environment is based on the Application Integration Architecture (AIA). In this context, *application integration* means users integrating with applications (via a common interface), applications integrating with the underlying system (via a rich set of services), and applications integrating with applications (via common methods of passing control and of passing and representing data).[1] Within Glacier, three components of AIA are used:

- DECwindows for graphical data display
- The Application Run-Time Utility Services (ARUS) for:
  - Memory allocation/deallocation
  - Condition handling
  - Conversion routines
  - Bottom of the stack condition handler
  - String mapping
  - String formatting
  - Process status
- The miscellaneous run-time library routines for:
  - Low-level math routines

[1] This definition is based on a memo authored by Scott G. Davis on 16-February-1988 titled "The Application Integration Architecture (AIA) Program".

— Common Multithread Architecture (CMA) routines

— Print System Model (PSM) routines

- Remote Procedure Calls (RPC), which are discussed in Section 50.4.4.4.3

50.4.4.1.2 Application Migration

To enhance and ease the migration of applications from other platforms such as VMS, ULTRIX, and POSIX, a set of run-time interface libraries may be used. These libraries implement a limited number of the system-specific interfaces on top of the Mica system services. Application developers will be encouraged to use the AIA libraries as the best means of achieving portability across the DIGITAL set of products. The use of the system-specific libraries will be discouraged.

Applications which use VAX/VMS system services are provided access to a restricted set of VMS service interfaces on Glacier. The exact list of services is to be determined.

Applications written using the ULTRIX or POSIX service interface may take advantage of the extensive Mica C Run-time Library interface as well as the Mica-POSIX interface.

50.4.4.1.3 Record Management Services

Mica RMS, together with the Application Integration Architecture (AIA), provides the highest user-level file and record access interface in the Glacier system. RMS is designed to promote the building of a common set of software components for PRISM/Mica and PRISM/ULTRIX. The interface to Mica RMS is designed to be a straightforward interface taking into account functionality of previous implementations. Through this interface and by direct use of Mica I/O subsystem functions, most of VMS RMS's functionality has been preserved.

The functionality of Mica RMS will be developed in stages across several releases. Key to the FRS requirements is transparent access of files through the local file system or through the distributed file system (DFS). Transparent access to standard input and output files is provided for as well.

50.4.4.2 Software Development Environment

The Glacier software development environment can be described in terms of the software which may be utilized by the application (see Section 50.4.4.1) and the tools available to the application developer.

Components of the common software architecture include:

- Calling standard and condition handling

  The PRISM calling standard specifies the instruction sequence and register conventions used when invoking a procedure. An interprocedure calling standard permits interoperability between procedures written in any conforming PRISM language. The calling standard has been optimized for the PRISM architecture, taking advantage of the large number of registers.

  Architected condition handling provides additional support for interlanguage operability, and a consistent and complete mechanism for program handling of hardware and software conditions arising during program execution.

- Object module and image file format

  A single architected format for object modules and image files supports building a single set of program development tools and language compilers across PRISM operating systems.

- Common Multithread Architecture

  The Common Multithread Architecture (CMA) provides applications and compiler-generated code with a consistent mechanism for implementing multithreading independently of the underlying hardware/software platform.

50.4.4.2.1 Program Development Tools

Program development tools for Glacier include:

- Linker

  The PRISM linker binds a collection of PRISM object modules into an image file for execution on the PRISM hardware. The linker design is closely integrated with the PRISM calling standard, the language compilers, and the PRISM software platforms to provide efficient, high-performance run-time image setup and execution.

- Librarian

  The PRISM librarian implements libraries of modules, such as object modules. Object module libraries increase programmer productivity and link-time performance by collecting many PRISM object modules into a single indexed file.

- Language compilers

  Glacier supports a full complement of programming languages:

  — PRISM FORTRAN provides a VAX-compatible FORTRAN language, implementing vectorization and parallel decomposition.

  — PRISM C is an ANSI-X3J11-compliant implementation with VAX C extensions.

  — Pillar, the PRISM systems implementation language, is supported for customer use, and is used within DIGITAL for PRISM layered product development.

  — VAX ADA will be ported to PRISM as a post-FRS product.

  — Pascal will likely be post-FRS product.

  — LISP is also likely to be a post-FRS product.

- Language Sensitive Editor (LSE) and Source Code Analyzer (SCA)

  LSE and SCA provide support for writing PRISM-based applications. LSE and SCA execute in the client system environment, enhancing application developers' ability to build programs for Glacier. LSE utilizes the Glacier client system interfaces to invoke the appropriate language compiler.

  PRISM language compilers optionally generate SCA files. SCA, which is integrated with LSE, provides developers with the ability to quickly locate usage of variables within large applications, thereby increasing programmer productivity.

- DEBUG

  PRISM DEBUG provides application developers with a highly interactive, language-oriented debugging tool. A DEBUG kernel executes along with the Glacier application being debugged. The DEBUG user interface executes on the client system, interacting with the DEBUG kernel via RPC. The PRISM DEBUG user interface is compatible with the client system debugger. For FRS, a VAX/VMS-compatible user interface is provided.

- Performance and Coverage Analyzer (PCA)

  The PRISM PCA is a tool that the application developer can use to analyze the run-time behavior of an application. PCA executes with the application on the PRISM system, writing the collected data to a disk file. Later, the PCA analyzer, running on the client system, can be used to analyze the data to pinpoint program bottlenecks, or determine program execution coverage.

50.4.4.3 System Management

The interface to Glacier system management executes on the client system or on Glacier, through the system console terminal. In either case, the system management user interface uses RPC to request the system management server, running on Glacier, to execute the specified management functions.

At initial system installation, the system manager specifies the password required for access to system management functions on the console terminal. In addition, the system manager defines which users on Glacier client systems throughout the network are permitted to execute system management functions remotely.

50.4.4.3.1 System Management Server

The Glacier system management user interface, executing on the client system, interacts with the system management server executing on Glacier to complete the requested management commands. In addition, the system management server is responsible for:

- Maintenance of the user authorization database

  The system management server is solely responsible for creation and maintenance of the on-disk user authorization database. No other process within the system directly accesses the on-disk database.

- Retrieving user authorization information for Glacier-based system software.

  System software on Glacier utilizes RPC to call the system management server when authorization information must be obtained.

50.4.4.3.2 Performance Monitor

Another important component of system management is the ability to monitor the performance of the Glacier system. The performance monitor user interface, integrated with the system management user interface, connects to the performance monitor server on Glacier to gather the performance information.

The performance monitor permits the system manager to view key system performance statistics and provides some capabilities for diagnosing potential system performance bottlenecks.

The performance monitor utilizes standard Mica operating system interfaces to gather performance information. The server is capable of supporting multiple users monitoring system performance concurrently.

50.4.4.3.3 Console Support

The Glacier console terminal interface can be used, rather than a client system, for execution of system management functions, diagnostics, or the User-level System Exerciser (USE). Access to these functions is password protected.

50.4.4.3.4 System Dump Analyzer

The system dump analyzer is utilized by DIGITAL support personnel and engineering to determine the cause of system failures.

50.4.4.3.5 Error Logging

All hardware errors are logged to an on-disk error log file. An error log display utility permits system management and DIGITAL support personnel to examine the error log contents.

50.4.4.4 Underlying Software Mechanisms

The following sections describe the underlying software mechanisms of Glacier, including the Mica operating system and networking mechanisms.

50.4.4.4.1 Mica Operating System

Glacier is built around hardware which implements the PRISM architecture and a base operating system which implements the Mica design.

Mica has been designed as an object-oriented system with a rich set of services available from user mode (note that most of these services will eventually be accessed via the Application Integration Architecture library). It is designed to support both the 32- and 64-bit PRISM architectures and should require minimal effort to move from a 32-bit-only implementation to a 32/64-bit implementation when 64-bit systems become available. The key points in the design are:

- Priority-based preemptive scheduling with provision for class scheduling
- A flexible memory management system which supports all allowed PRISM memory management implementations
- Multiple threads of execution within a single address space
- A layered I/O architecture for the support of physical devices, file systems, and concepts, such as volume shadowing, volume striping, virtual terminals, and so on
- A centralized ACL-based security architecture for all objects
- Protected subsystems, that is, user processes which act as servers with amplified security profiles on behalf of client processes, charging back resource usage to those clients
- An implementation written almost entirely in the Pillar language, which provides block structure, strong typing, and structured condition handling, and has been designed as a portable system implementation language

50.4.4.4.2 DECnet-Mica Phase V

The client/server nature of Glacier requires strong network support. An implementation of the DIGITAL Network Architecture (DNA), Phase V is used to provide the required network services. The components of DNA that are included in the FRS product are:

- DNA naming services
- Network communication services
- Network management
- Network event-logging server

50.4.4.4.3 Remote Procedure Calls

As mentioned earlier, a remote procedure call (RPC) facility is used throughout Glacier for interprocess communication. This communication may be with off-node entities, such as the client context server, and with local node entities (protected subsystems), such as the system management server.

Mica RPC comprises two main components: an RPC stub generator, and RPC run-time facility. The RPC stub generator provides a language interface to the RPC run-time facility. Stubs are written in a high-level language similar to Pillar, which is the implementation language for Glacier software. Stubs are simply the local procedure definition of the procedure being called remotely. Stubs are easy to write and they insulate the user from the underlying RPC run-time facility and transport mechanism. The stub generator is used in developing the Mica system and is not intended to be included in the product.

The Mica RPC run-time facility provides the mechanisms used by the stubs. These include data marshaling, multiple context maintenance and client/server binding.

50.4.4.4.4 Distributed File Services (DFS)

The Distributed File Services for Glacier comprise three main components:

- Disk services for accessing client disks from a Glacier server
- Disk services for accessing Glacier disks from a client system
- Management services for administering DFS remotely

For FRS, DFS Version 1 protocol is supported. This protocol enables systems with ODS–II-based file systems to interoperate. In future versions, heterogeneous file system interoperation may be allowed.

50.4.4.4.5 Job Controller Server

The focal point for application activation requests are serviced by a Glacier-based server, called the *job controller server* (JCS). The JCS services requests from the client context server to activate, suspend, resume, and debug the application programs. The JCS also provides the CCS with the application's exit status information.

50.5 Special Challenges

The Glacier product is a computing element, specifically a compute server, designed to provide cost-effective computational resources to an array of workstations. The compute server environment presents several challenges that were not addressed by previous computing environments. Those challenges are:

- Seamless client integration
- Remote management

Integration of the compute server with a group of heterogeneous client workstations is essential. Workstations, like other computer systems, differ in functionality, cost, and performance. It is expected that a workstation-based network will have several different stations requiring various levels of service by a range of users. Users must be able to access Glacier from each workstation in a manner similar to the familiar workstation environment.

Glacier's lack of a user interface requires that system management be done remotely. Glacier must require only minimal interaction from the console terminal before becoming available as a remotely managed system.

50.6 Outstanding Issues

- The support of ULTRIX client systems has a number of unknowns associated with it. As mentioned earlier, VAX RPC supports only VMS systems today. Secondly, the DFS Version 1 protocol supports only ODS–II file systems. The Version 2 protocol may include support for the ULTRIX file system. An alternative to DFS for ULTRIX is NFS. This would require that NFS be implemented for Mica.

  These are only a couple of the open ULTRIX client issues. A preliminary ULTRIX client support chapter outlines the issue more completely. In any case, ULTRIX support is planned for shipment after VMS client support.

CHAPTER 51

MICA COMPUTE SERVER SUPPORT

51.1 Overview

This chapter describes the software support required on Mica to allow a user on VAX/VMS or VAX/ULTRIX to execute or debug a program on the Mica under a tightly coupled model.

Under this compute server model, there is software on the client system called the Client Context Server. The Mica support software and the Client Context Server work together to enable the execution of programs on the Mica system. The Client Context Server supports the compute server job on the Mica system by providing client system services that can not or should not be performed on the Mica system itself.

The tightly coupled model is designed for large compute-bound application programs. These are single stream monolithic programs written in high-level languages. They operate in user mode and use few system services or privileged functions. For example, these programs would not create or delete processes or modify user privileges. The user on the client system is provided access to Mica-based compilers, a linker and a debugger to produce the Mica image and a mechanism to execute the Mica image on the compute server. The development environment is that of the client system.

Under this tightly coupled model, the compute server runs the Mica image on behalf of the client system. The environment of the Mica image on the compute server is that of the client system. All I/O from SYS$INPUT (stdin) and to SYS$OUTPUT, SYS$ERROR (stdout, stderr) is handled by Remote Procedure Calls (RPCs) to the Client Context Server. The Client Context Server then performs the I/O on the client system. The image appears to be running on the client system, reading from standard input and writing to standard output/error. Use of the Client Context Server ensures that the compute server job interacts properly with other components of the client operating system, such as redirected files, pipes, and sockets. Logical names, environment variables, networking visibility, and process visibility are taken from the client system.

51.1.1 Goals

The Mica support for the compute server product has these basic goals:

- Provide the Mica portion of the mechanism for activating a Mica image from the client system.
- Provide the Application Integration Architecture (AIA) environment for the compute server programs.
- Provide support for development tools including the Debugger.

51.1.1.1 Activation of a Mica Image

Mica provides a mechanism for the client system to activate Mica images on the compute server. This mechanism is independent of the client operating system and notifies the client system of the termination of the compute server image. The image is activated on behalf of the client system and is considered an extension of the client's computing resources. All files that are available on the client are available to the compute server. This is accomplished via Mica RMS, which handles file parsing and uses the Distributed File System (DFS) to access files. The environment of the image (logical names, shell environment variables, process permanent files, command line information) on the compute server is that of the Client Context Server running on the client system. Termination of either the user's process on the client or a Mica compute server image means the termination of the other.

\ It is unknown at this time if DFS will support the ULTRIX file system. \

51.1.1.2 Application Integration Architecture

The AIA on the compute server provides a system-independent representation of system services and the DECwindows user interface. In the AIA environment, the user does not see the Mica system service calls or have access to them. AIA calls are carried out in one of three ways:

- An AIA call may be carried out solely on Mica via system service calls.
- An AIA call may result in an RPC to the Client Context Server to carry out the AIA call solely on the client.
- An AIA call may result in a Remote Procedure Call (RPC) to the client for client context information and then make additional calls to Mica system services.

51.1.1.3 Support for Development Tools

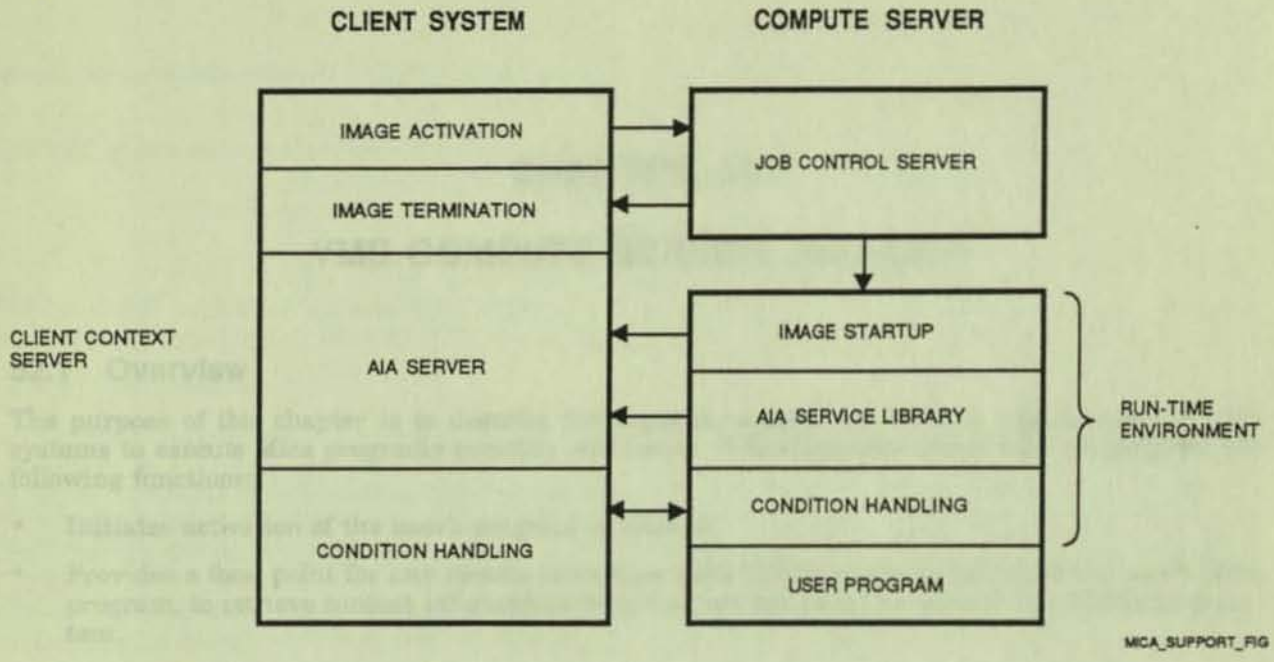
The Mica software will provide support for the SDT development tools. This includes the SDT debugger.

\ The support that is to be built into Mica is TBD.\

51.1.2 Components

The tightly coupled client/server model consists of several components as shown in Figure 51–1.

Figure 51–1: Mica Compute Server Support



51.1.2.1 Mica Components

The Mica compute server support consists of two major parts: the Job Control Server and the Run Time Environment.

The Job Control Server handles requests from the Client Context Server to execute a Mica image. Upon receiving an RPC to start a compute server image, the Job Control Server creates a thread that acts as a monitor for the compute server job. This new thread authenticates the client's request and creates the compute server job. It then waits for the compute server job to terminate. There is a one-to-one mapping between compute server jobs and Job Control Server threads. Once the compute server job terminates, the Job Control Server thread performs an RPC to the Client Context Server to convey the exit status and accounting information.

\ The thread per job design is required to handle status objects. Fewer threads may be required given a different mechanism for handling status return, for example by using a message FPU for status information. \

The Run Time Environment in Mica for the Compute Server consists of three parts:

- Image Startup Procedure—This procedure is called before the user code is executed. It sets up all handlers and general environment. The procedure is passed enough information to identify the corresponding Client Context Server. This procedure initiates the first RPC to the Client Context Server, which returns the DFS specification of the default directory and process permanent files.
- AIA Service Library—This library contains the AIA routines available in the Mica environment.
- Error and Condition Handling Routines—These routines communicate conditions and errors to and from the Mica image and the Client Context Server. If a software error or interrupt occurs during the execution of the Mica image, these routines inform the Client Context Server via an RPC. If the Client Context Server receives an interrupt, an RPC is sent to the Mica image to inform it of the condition.

CHAPTER 52

VMS COMPUTE SERVER SUPPORT

52.1 Overview

The purpose of this chapter is to describe the support required on VMS to enable users of VMS systems to execute Mica programs remotely on Glacier. VMS compute server support performs the following functions:

- Initiates activation of the user's program on Glacier.
- Provides a focal point for any remote procedure calls (RPCs) made on behalf of the user's Mica program, to retrieve context information from the corresponding process on the VMS client system.
- Handles notification of any errors that might occur during program execution on Glacier.
- Handles notification of the Mica image termination, and performs any required cleanup.

52.1.1 Requirements

Glacier is targeted toward two types of application programs at FRS: single stream high-level language applications, and utilities written by Digital. Support must be provided which enables a VMS user to execute these programs and utilities on Glacier. Little or no modification to user-written programs should be required, except for those that use VMS system routines not supported on Mica.

The program execution environment must appear "seamless" to the VMS user. "Seamless" means, at a minimum, the following:

- A program is invoked through some variant of the RUN command issued by the user on the local VMS system. (For utilities written by Digital, a program is invoked using other DCL commands.)
- Terminal dialogue with the user appears identical to the dialogue that occurs when the program is executed locally on the VMS system, with the exception of DCL CLI calls.
- Termination of execution is indicated to the user.
- Output from the program is identical to the output that results from execution on the VMS system.
- The user has the capability to interrupt and/or terminate execution (through the CTRL/Y mechanism), just as when the program is run locally.

VMS compute server support must provide access to a limited number of VMS system routines through the RPC mechanism. In particular, support must be provided which enables the compute server to access devices that are not directly attached to the Mica system and are not served by the Distributed File Service. These devices, instead, are attached to the user's VMS system, such as terminals.

The stated aim of the Mica programming environment is to support applications that adhere to the principles of AIA. VMS compute server support must provide distributed support for some Mica non-local AIA services.

52.1.2 Assumptions

The following assumptions have been made:

- That the VMS system and the Mica system have compatible RPC implementations which are both efficient and reliable.
- That a user identification and rights list exist on Mica for each user; it is not necessary that the user quotas on both the VMS system and the Mica system be identical.

52.1.3 Functional Description

Compute server support under VMS is provided by a single VMS executable image. When a VMS user requests execution of a Mica image, the VMS compute server support image runs in the context of the user's VMS process.

52.1.3.1 Image Activation

VMS compute server support software initiates activation of the user's Mica program by using an RPC call to a job controller server that resides on Mica. The Mica job controller server effects the image activation of the Mica program, and returns the status of image startup to the VMS support software. A successful completion of the image activation call puts the VMS compute server software into an RPC wait state, as a client context server. The client context server can emerge from an RPC wait state at any time to handle RPC calls from the executing Mica program.

52.1.3.2 RPC Calls for VMS Services

The VMS client context server services RPC calls that occur during execution of the user's Mica program. The requests to be serviced result from either an AIA service call or from an RMS request to perform I/O to a device on the user's VMS system. The client context server executes the required services and returns the results to the user's program. Examples of VMS performed services include I/O from and to the user's terminal and calls for user-specific context information such as logical name translation.

52.1.3.3 Condition Handling

Conditions that occur during the program execution on Mica cause notification to be sent to the VMS client context server, which in turn notifies the user of execution status. If either partner (on VMS or Mica) terminates, the RPC run-time system provides notification to the remaining partner, which will exit. If the user interrupts program execution through a CTRL/Y, the Mica program will be notified to exit.

52.1.3.4 Termination

Normal program termination is reported to the VMS client context server, which passes exit status to the user. Any RPC bindings are broken down by the VMS client context server at termination.

52.1.3.5 Debugger Support

The VMS client context server provides support for the VAX/PRISM Distributed Debugger.

CHAPTER 53

ULTRIX COMPUTE SERVER SUPPORT

53.1 Overview

This chapter describes the software support that enables a user on ULTRIX to execute or debug a program on a Glacier compute server. A Glacier compute server can be connected to one or more ULTRIX systems to provide high-performance computation capabilities. The compute server is an extension of the ULTRIX client computing environment. The compute server operates with its client in a "seamless" fashion. The client support software on ULTRIX interacts with software on Glacier to execute Mica images on the Glacier system on behalf of the client users. The images appear to be executing on the client system, and the client users are unaware that the images are actually executing on Mica.

53.1.1 Goals

The ULTRIX software support for the compute server product has these basic goals:

- Provide the mechanism on ULTRIX to allow users to execute a Mica image on Glacier in a "seamless" fashion.
- Provide access to the client environment for compute server programs.
- Provide support for Mica development tools on ULTRIX.

53.1.1.1 Execution of a Mica Image

The ULTRIX compute server support software provides a "seamless" mechanism for executing a Mica image on Glacier. To execute a Mica image an ULTRIX user should simply invoke a user command just as they would execute an image locally on the ULTRIX client system. The activities of the ULTRIX compute server support software must be transparent. The user can redirect standard input, output and error to/from files or ULTRIX pipes to other commands. The user can invoke a Mica compute server program from a shell procedure.

53.1.1.2 Access to the Client Environment.

The ULTRIX compute server support software must provide access to the client environment for the compute server program. This support includes:

- Communication with other processes via ULTRIX pipes, named pipes (FIFO), or sockets.
- Handle all remote procedure calls (RPC) from the compute server image that are the result of an Application Integration Architecture (AIA) call.
- Performing I/O to/from standard input, output and error.

This compute server model is designed to handle large, compute-bound application programs. These applications are written in high-level languages and:

- Use few system services.

- Do not fork/exec other processes.
- Do not use shared memory for inter-process communication.

53.1.1.3 Development Tool Support

All program development tools for the compute server must run under the tightly-coupled model. They should be invoked using the same syntax as any other ULTRIX development tool. The names of these tools will be different from the standard ULTRIX tools to avoid naming conflicts.

The client compute server support software for these development tools will handle RPCs resulting from AIA Command Language Interpreter (CLI) calls. These RPCs will take the ULTRIX command line arguments and interpret them for the tool running on the compute server.

53.1.2 Functional Description

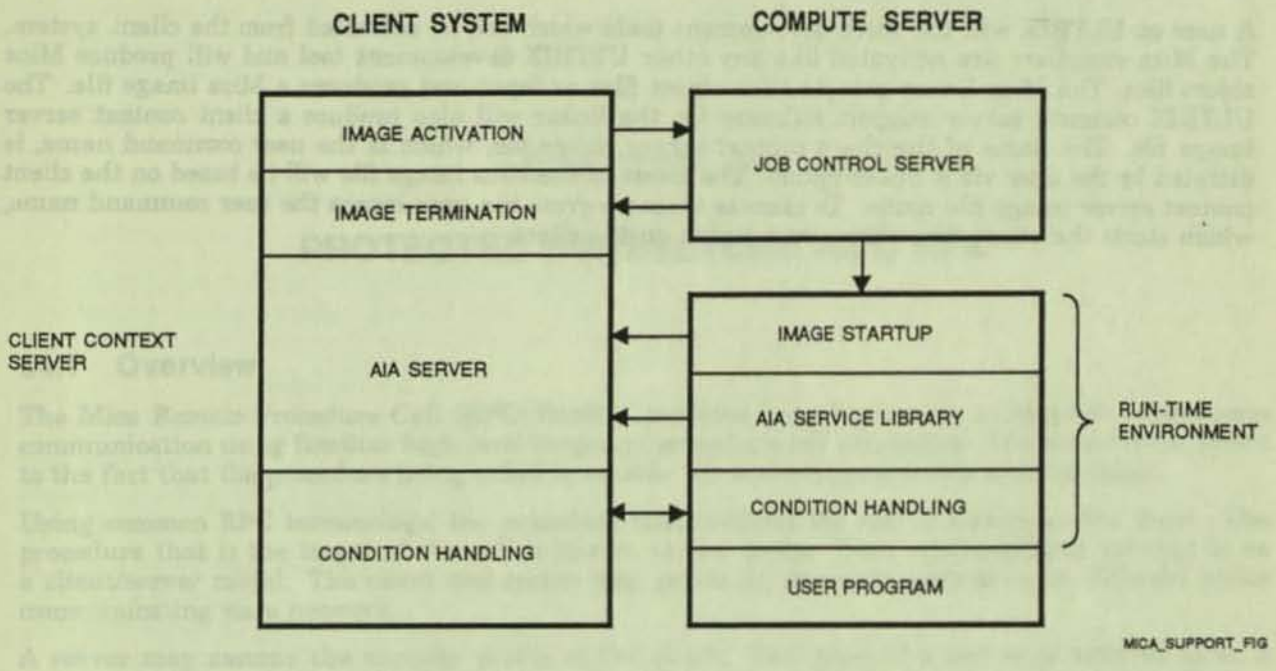
The tightly-coupled client/server model consists of several components as shown in Figure 53–1. There are three major components to this compute server model:

- Job control server—The job control server is a registered server running on the compute server.
- Client context server—The client context server is the compute server support software that runs on the ULTRIX system when the user executes the user command.
- User's program— The user's program is the Mica image that will be activated by the job control server.

The ULTRIX user issues a user command that creates a child process to the shell and executes the client context server code. The process registers itself with the remote procedure call (RPC) binder as a server. The process then sends an RPC to the job control server, requesting activation of the compute server image on Glacier. When this RPC returns with a successful status the compute server image has been started. The client context server then loops on an RPC wait, and comes out of the wait state to handle any RPC requests from the compute server image, including its eventual termination.

\ The exact sequence of the initial RPC and binding request are TBD but we have the requirement that there should be no window of vulnerability where a binding does not exist. \

Figure 53–1: ULTRIX Compute Server Support



53.1.2.1 The Client Context Server

The ULTRIX compute server support software, called the client context server, consists of four parts:

- Image Activation—The first task of the client context server is to activate the Mica image on the compute server. This is done by sending an RPC to the job control server on the appropriate Glacier system. The RPC returns to the client context server with a status. The status indicates whether the compute server image has been started successfully or not. If the client context server can not start the compute server image successfully it displays a message to the user and terminates.
- Image Termination—When the compute server image terminates the client context server receives an RPC containing the compute server image's exit status. The client context server then exits with this status.
- AIA Server—The ULTRIX client context server handles RPCs from the compute server image. The request to be serviced results from either an AIA service call or from an RMS request to perform I/O to a special file on the user's ULTRIX system. The client context server executes the appropriate service and returns the results to the compute server image. The ULTRIX client context server translates RMS RPC request from the compute server image by calling ULTRIX system calls.
- Condition Handling—Interrupts and error conditions that occur during the execution of a Mica image on the compute server are communicated to the ULTRIX client context server. If either partner (on ULTRIX or Mica) terminates, the RPC run-time system provides notification to the remaining partner, which exits. If the ULTRIX client context server receives a signal it will terminate and thereby causing the termination of the Mica compute server job.

53.1.2.2 Mica Program Development on ULTRIX

A user on ULTRIX will use Mica development tools which can be activated from the client system. The Mica compilers are activated like any other ULTRIX development tool and will produce Mica object files. The Mica linker accepts Mica object files as input and produces a Mica image file. The ULTRIX compute server support software for the linker will also produce a client context server image file. The name of the client context server image file, which is the user command name, is dictated by the user via a linker option. The name of the Mica image file will be based on the client context server image file name. To execute their program the user issues the user command name, which starts the client context server running on the client.

CHAPTER 54

PROTECTED SUBSYSTEMS AND RPC

54.1 Overview

The Mica Remote Procedure Call (RPC) Facility provides a mechanism to accomplish interprocess communication using familiar high-level language procedure call semantics. The term *remote* refers to the fact that the procedure being called is outside the current procedure's address space.

Using common RPC terminology, the procedure that initiates the call is known as the client. The procedure that is the target of the call is known as the server. This relationship is referred to as a client/server model. The client and server may reside on the same system or on different nodes communicating via a network.

A server may assume the security profile of the client. This type of a server is referred to as a *protected subsystem*. A complete description of how security profiles are assumed is found in Chapter 10, Security and Privileges.

There are two key components to the Mica RPC facility: a stub generator and the run-time facility. The stub generator is used to describe a procedure's interface. For each interface, a client and server stub is created that hides the communication between client and server. A complete description of the stub generator is found in Chapter 55, RPC Stub Generator. The Mica RPC run-time facility provides a high-level interface to the communication transport mechanism. Its interface defines the types of messages needed to invoke and pass arguments to a procedure in a remote environment. This chapter focuses on the Mica RPC run-time facility.

Figure 54–1 illustrates the relationship between the various interfaces used during an RPC call. This model depicts the current DNA RPC architecture. The highlighted region of the illustration is the focus of this chapter.

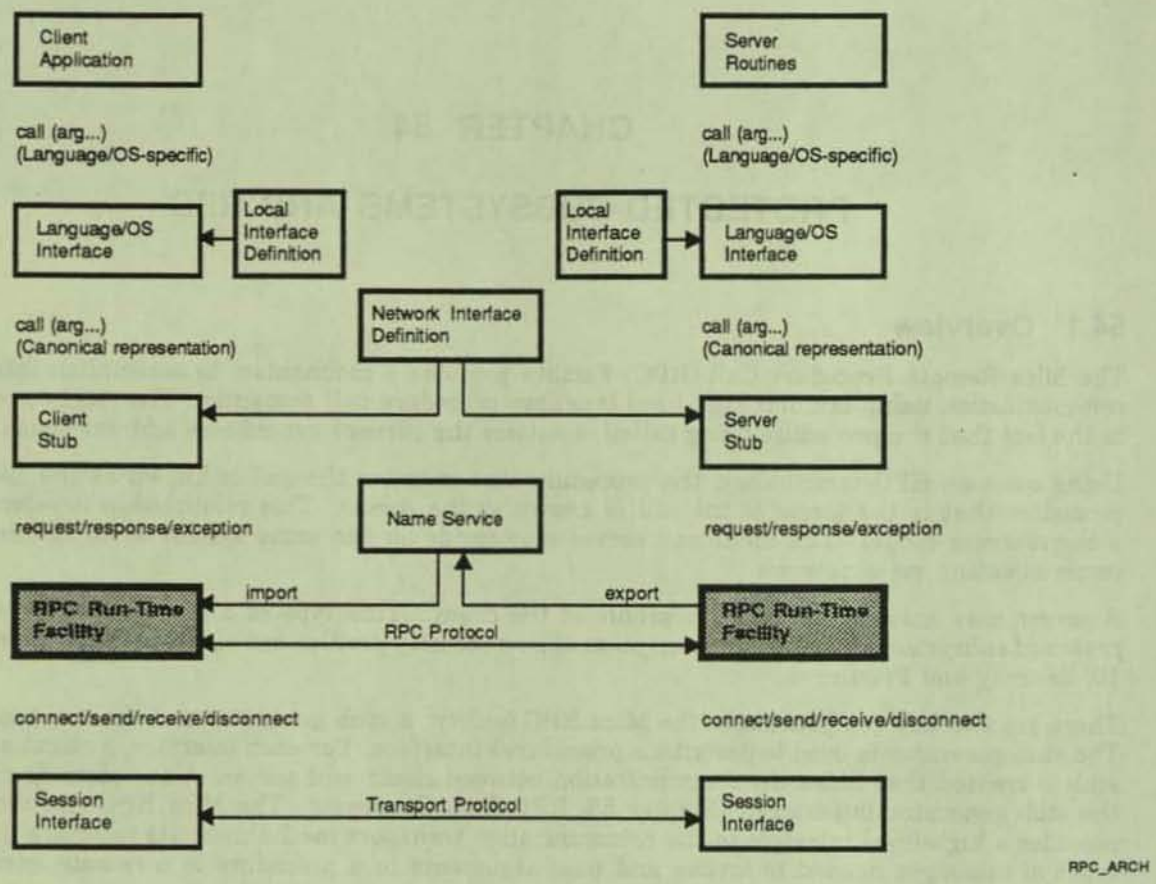
54.1.1 Goals

The RPC mechanism is used extensively throughout Mica. In fact, the nature of Glacier is derived from its RPC interface. For example, a program is started on Glacier is through an RPC call from the client system.

The Mica RPC run-time facility is designed with the following goals:

- Provide the RPC functionality required by Mica system components
- Provide a functional basis for protected subsystems
- Provide an easy migration path to corporate RPC
- Provide RPC functionality that allows most usage to be hidden by the stub generator

Figure 54–1: The RPC Architecture Model



54.1.1.1 Functionality for Mica System Components

The concept of RPC seems simple. However, the actual implementation of an RPC mechanism that provides the same level of functionality found in standard local procedure calls is quite complex. The level of complexity is increased further when the client and server are running on heterogeneous operating systems.

Mica system components require the following RPC functionality:

- Interoperation—The RPC mechanism used by Mica is designed to interoperate with VAX/VMS, Ultrix, and Mica.
- Procedure call behavior—By default, an RPC behaves like a procedure call. That is, when the server routine returns, it is finished. The server routine is guaranteed to be called only once per client call. Any deviation from this (for example, idempotent and stream calls), is a special case and does not compromise the true procedure call semantics.
- Clients as servers—A client may act as a server to another client. A client can obtain a binding to itself for a specific interface, and would pass that binding in a call to a server. The server may then use that binding to make RPC calls back to the client.
- Servers as clients—A server may act as a client to another server.

- Binding—The binding service is used by a client to obtain information required to communicate with a server. At FRS, Glacier requires limited automatic binding and naming services. The ability to provide run-time binding information without changing the call stub is done using logical names.
- Context—Context may be maintained across server calls through the use of context handles. A context handle is an opaque datatype that is passed on each call to a server to identify a specific context. The scope of the context handle is an instance of the server. This allows authentication, file context, and other context to be maintained across server calls and significantly increases performance.
- Disconnect notification—Notification of failure is provided to a server if the client holding an opaque context handle dies or disconnects, to the caller if the server dies or disconnects during a call, and to a server if the caller dies during a call.

54.1.1.2 Functional Basis for Protected Subsystems

The Mica RPC mechanism is used extensively to call normal servers and protected subsystems. A protected subsystem differs from a normal server in two key areas: the RPC protocol transport mechanism and client impersonation.

The performance of the communication transport between a local client and a protected subsystem must be extremely good. It appears that DECnet does not provide sufficient performance for local protected subsystem calls. A function processor providing local virtual circuit functionality provides an alternative transport path. The RPC mechanism will select the appropriate transport mechanism given the locality of the server.

A protected subsystem receives the identifiers of a client so that it may service the request using the identity of the client. The access rights of any client are based on local authentication and identifiers. Mica provides a set of impersonation services that are used by the protected subsystem to manipulate its own identity.

54.1.1.3 Easy Migration to Corporate RPC

An aggressive corporate RPC architecture plan is being pursued. If a corporate architecture is not available, then Mica RPC will be designed to allow for migration to the corporate version at some later date.

\The first step in building Mica RPC will be to interoperate with VAX RPC. Should a DNA RPC architecture not be approved, VAX RPC can meet all Mica RPC requirements and will be shipped with the client support software.\

54.1.1.4 Hide RPC Usage Behind the Stub Generator

The functionality provided by the Mica RPC run-time facility is invisible to most applications. That is, most applications call a remote procedure exactly as they would call a local procedure. This is accomplished by generating a stub procedure for each remote procedure. The stub procedure is simply a routine that converts a procedure call into a set of RPC run-time facility calls.

Besides simplicity of application development, the use of a robust stub generator has another key benefit. The stub generator isolates the application from the underlying RPC implementation. This allows protocols and other implementation details to undergo substantial change without affecting the higher level applications.

54.1.2 Nongoals

The Mica RPC nongoals are those items that will be deferred to a future release. The design of Mica RPC does not preclude their implementation at the appropriate time. The RPC run-time facility nongoals include the following:

- Provide a customer-visible RPC facility
- Provide an all-encompassing mechanism for inter- and intranode communication
- Provide RPC functionality that interoperates with other RPC protocols

54.1.2.1 Customer Visibility

The underlying Mica RPC facility will *not* be visible to external customers in the FRS product. Future releases will provide RPC stub generation and RPC run-time support for direct customer use.

54.1.2.2 All-Encompassing Mechanism

A robust RPC facility requires a significant number of features that are not being considered for FRS implementation on Mica. The following list describes some of the features that are commonly found in an RPC facility, but which will not be present in Mica RPC at FRS:

- Stream calls—A stream call is a call or callback that is queued to the procedure that executes the call. The client continues as soon as the call is queued. A stream call does not have output and cannot generate a condition or return status.
- Idempotent calls—An idempotent call allows the client to repeat a command until a response is received with the knowledge that the server's state is consistent even though the call was repeated.
- Condition handling—Conditions generated by a procedure may cause an action routine or handler in the caller's environment to be invoked.
- Call-back procedures—A client may provide call-back procedure arguments to the server. A new binding is not required for the server to call a callback via a procedure argument. \This functionality may be required by the DNA architecture.\
- Call interrupt—The interrupt message instructs the partner to abort processing a service request. \This functionality may be required by the DNA architecture.\
- Load balancing—The binding services may select the server to be used based on loading balance /performance algorithms.

54.1.2.3 Interoperation with other RPC protocols

The interoperation with other RPC protocols includes common data type representation, server binding, and condition handling. RPC interoperation could eventually occur with the following:

- PRISM 64-bit systems
- Significant workstations (for example, SUN and Apollo)
- Emerging industry standards (for example, OSI)

54.1.3 Communications Transport

The communications transport mechanism varies depending upon the relative location of the client and server. The Mica RPC mechanism is transport independent. It assumes that the transport provides logical links for reliable message transport.

For FRS, two transport mechanisms are available. The DECnet session interface is used for internode communication. Intranode communication is accomplished through either the DECnet session function processor or a local transport function processor with a session-like interface.

54.1.4 Issues

The Glacier field test is scheduled to begin September 1989. In order for DECwest to implement and make use of an RPC mechanism that conforms to the corporate architecture, an architecture that meets the basic requirements must be in place no later than March 1988.

CHAPTER 55

RPC STUB GENERATOR

55.1 Overview

The Mica RPC Stub Generator provides transparent access to remote procedure calls.

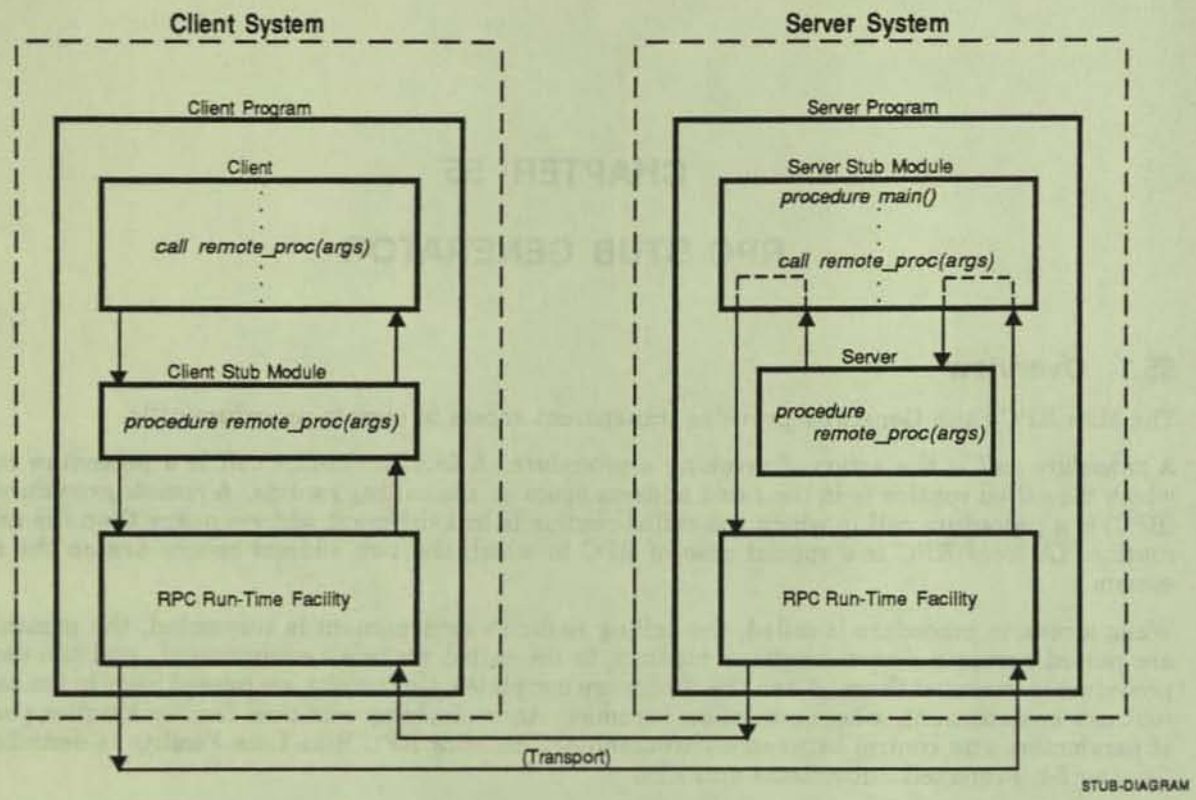
A *procedure call* is the action of invoking a procedure. A *local procedure call* is a procedure call in which the called routine is in the same address space as the calling routine. A *remote procedure call* (RPC) is a procedure call in which the called routine is in a different address space than the calling routine. (A *local RPC* is a special case of RPC in which the two address spaces are on the same system.)

When a remote procedure is called, the calling routine's environment is suspended, the parameters are passed across a communications medium to the called routine's environment, and the desired procedure is executed there. When the procedure completes, the results are passed back to the calling routine's environment, where execution resumes. An underlying run-time facility handles passing of parameters and control between environments. The Mica RPC Run-Time Facility is described in Chapter 54, Protected Subsystems and RPC.

A remote procedure call can be made to look and behave almost exactly the same as a local procedure call. The Mica RPC Stub Generator does this by hiding the differences in *stub modules*. When the calling program (the *client*) calls what it thinks is a local procedure, it actually calls a *client stub* routine in the client stub module. The client stub routes the call to the *server* using the Mica RPC Run-Time Facility. A *server stub* routine in the server stub module receives the call and makes the call to the real server procedure. To the server procedure, it appears as though the server procedure had been called locally.

Figure 55–1 shows the flow of a typical remote procedure call.

Figure 55–1: The Flow of a Remote Procedure Call



In practice, the client and the server may need to be aware that they are operating in an RPC environment. The use of stubs cannot eliminate some essential differences between local and remote procedure calls. In a distributed environment, the client and server can fail independently, performance can be quite different, and there is no shared address space. Some applications may want a specific server to process their calls, or they may want to communicate with a number of similar servers.

The Mica RPC Stub Generator can operate in a semi-transparent mode that allows the client and the server to use some of the capabilities of the underlying RPC mechanism. For example, the client can specify which server is to execute a remote procedure by supplying a *binding* argument on the call, and the server can ask to be informed of client termination if it needs to clean up client-specific context. Clients and servers must call the Mica RPC Run-Time Facility directly when they need to use a capability to which the stub generator does not give them access.

55.1.1 Requirements, Goals, and Nongoals

This section outlines the requirements, goals, and nongoals of the Mica RPC Stub Generator. Requirements are those attributes the stub generator must have; goals are those attributes the stub generator should have, but may not completely satisfy; nongoals are those attributes which the stub generator does *not* have.

55.1.1.1 Requirements

The stub generator must support user-supplied server initialization and termination routines.

The stub generator must support all argument data types and all argument-passing mechanisms supported by the Mica RPC Run-Time Facility.

The stub generator must support the following items, if they are supported by the Mica RPC Run-Time Facility:

- Server-maintained, client-specific context—This includes notifying the server of client termination when such context is being maintained.
- Calls from the server to procedures in the client—These *callbacks* can only be made by a server when it is executing a call from the client it is calling back.
- Streamed calls—These calls allow the client to resume execution immediately when the call has been sent to the server, without waiting for the server to complete execution of the call.
- Multithreaded applications—This is done using binding arguments.
- Call interruption.

55.1.1.2 Goals

The stub generator should support the following items, if they are supported by the Mica RPC Run-Time Facility:

- Raising of conditions in the caller's environment as a result of unhandled conditions in the called routine's environment.
- Version control.
- Call ID parameter on all procedure calls, if requested—The call ID is used to provide support for client authentication.

55.1.1.3 Nongoals

At FRS, the Mica RPC Run-Time Facility will not be available to user-written applications. Thus there is no need for the stub generator to be shipped with Mica; it will instead be used as a tool for the development of internal applications that need RPC. These applications include Monitor, system management, the client/server interface software, and various components of AIA. The long-term strategy for the stub generator is discussed in Section 55.1.5.

The stub generator does not support asynchronous remote procedure calls.

The stub generator does not produce language-specific header modules. A header module contains a source-language representation of the data types and procedures defined in the package. Instead of producing header modules, the stub generator produces definition modules, as described in Section 55.1.2.

55.1.2 Operation of the Stub Generator

The stub generator takes as input a *package definition* written in *Stub*, the *package definition language*. Stub is based on Pillar. Language elements required for package definition are added, and elements of Pillar that do not make sense in a package definition are deleted. In addition, the use of certain language elements is restricted. This prevents the use of data types and argument-passing mechanisms that are not supported by the Mica RPC Run-Time Facility.

The stub generator produces as output the Pillar source modules described in the following table:

Module	Contents
Definition module	This module contains definitions of the data types and procedures defined in the package. The definition module also defines the *package definition block*, a global record used by the Mica RPC Run-Time Facility. The compiled version of the definition module may be imported by client programs and server implementations that are written in a language whose compiler accepts definition modules.
Client stub module	This module contains client stubs for each procedure implemented in the server. The client stub module imports the definition module.
Server stub module	This module contains the server stub for a package. The server stub, which is the main entry point of the server image, calls server procedures as remote procedure calls are received from the Mica RPC Run-Time Facility. The server stub module imports the definition module.

55.1.3 Implementation Strategy

The Mica RPC Stub Generator is implemented using the compiler shell and super shell developed by the DECwest compiler group. As their names imply, these shells are designed as a framework around which various compilers can be built. They provide routines for handling language-independent tasks that are common to the various compilers.

The following table explains the various components of the stub generator. (In the table, "host" refers to the system on which the stub generator runs, and "target" refers to the system on which the compiled program runs.)

Component	Purpose	Dependencies
Language driver (xLD)	Command line parsing and opening and closing language-specific files. Main entry point for stub generator image. Calls *CS Master* (see below) to begin compilation.	Language and host OS
Super shell (SS)	I/O, memory management, and error handling. Interface between host operating system and remainder of the stub generator.	Host OS
Compiler shell (CS)	Common support routines, including the lexical analyzer. Also contains *CS Master* routine, which controls compilation by calling FE and BE.	None
Front end (FE)	Syntax and semantic analysis.	Language
Back end (BE)	Code generation.	Target RPC architecture

The super shell and the compiler shell are provided by the DECwest compiler group. The language driver, front end, and back end are part of the Mica RPC Stub Generator project. Parts of the language driver and the front end (notably, command line parsing) are based on their counterparts in the Pillar compiler.

The code generated by the back end is Pillar source code, as described in Section 55.1.2, rather than the object code normally produced by a compiler back end.

55.1.4 Dependencies

The stub generator depends on the stability of the compiler shell and the super shell. The shells exist as of this writing, but are subject to change as development of the Pillar and C compilers continues. The stub generator does *not* depend on the Pillar front end or back end.

The stub generator needs to know the details of the interface to the Mica RPC Run-Time Facility. These details are not currently available, but the stub generator is designed in a modular fashion so that it can easily adapt to various RPC mechanisms.

55.1.5 Long-Term Mica RPC Stub Generator Strategy

The stub generator described in this chapter is intended to be used as a tool for internal development of distributed applications. When the Mica RPC Run-Time Facility becomes available to users, the stub generator also needs to be available. The form that the user-visible stub generator takes is undecided.

The internal version of the stub generator produces Pillar source code. This source code needs to be separately compiled using the Pillar compiler. It is probably (but not certainly) desirable for the user-visible stub generator to produce object code directly. This could be accomplished by modifying the stub generator to use the back end developed by the DECwest compiler group.

The corporate RPC effort includes the work of a group in SDT that is defining the language requirements created by RPC. That group is also defining the required program development tools, which may or may not include a stub generator and an associated stub language. The Mica RPC Stub Generator is designed and implemented in a way that makes it relatively easy to adapt to the corporate RPC model, if necessary. Because it will not be user-visible at FRS, we will not have compatibility problems.

CHAPTER 56

AIA USER INTERFACE

56.1 Overview

This chapter describes the interface between Mica application programs and their human users. DECwindows provides the user interface when the user is at a workstation with a bitmap (graphics) terminal. This chapter describes the DECwindows implementation on Mica.

RMS provides the only character-cell terminal support in the initial release of Mica. RMS supports only a simple interface to these terminals; essentially, an application can only read and write line-oriented data. See Chapter 26, Record Management Services, for a description of RMS's support for character-cell terminals.

56.1.1 Goals

Because of Mica's compute-server nature, Mica applications and their users will be on separate systems. The AIA User Interface must provide support for this separation.

A state-of-the-art user interface must be provided for workstation users. For users at character-cell terminals, only a simple interface is required.

56.1.2 DECwindows

DECwindows[1] is built upon the industry-standard X Window System Version 11™ to give workstation users a network-transparent application programming environment for windowing, graphics, and state-of-the-art user interface services. When coupled with a base set of DIGITAL-developed core applications and a library of third-party applications, DECwindows provides DIGITAL's customers with a single, consistent view of application development and user interfaces.

By implementing DECwindows on Mica, we allow applications running on Mica to communicate with users on remote workstations. These remote workstations do not necessarily have to be direct clients of the compute server, as long as they are part of the compute server's DECnet network. A user on a nonclient workstation must access a client system via DECnet in order to get the application running on the compute server, but the compute server can communicate directly back to the nonclient workstation.

A full implementation of DECwindows consists of the following components:

- The X Window System (including device support)
- Application programming libraries. For example, the DECtoolkit
- The User environment. For example, the window manager
- Core applications. For example, the EPIC/WRITER

[1] The descriptions of DECwindows and its components in this section are to a large extent extracted and adapted from Peter George's *VMS DECwindows Version 1.0 Project Plan*.

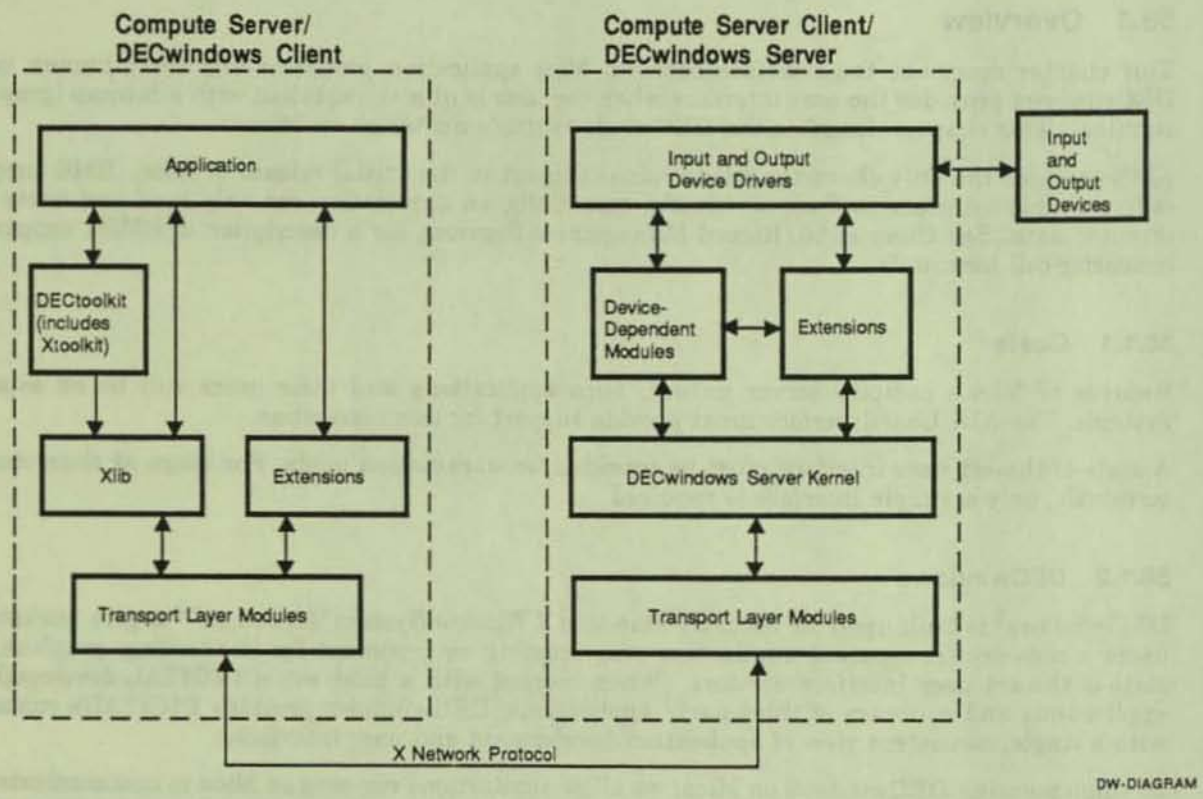
™ X Window System, Version 11 is a trademark of the Massachusetts Institute of Technology.

Figure 56–1 shows the major components of a DECwindows implementation, with the various components listed above broken into their subcomponents[2] . These subcomponents are described in the following sections.

NOTE

Because of Mica's compute server nature, its implementation of DECwindows does not include the device support and user environment components. These components execute as part of the compute server client's DECwindows software. In addition, we have no plans to provide any core applications. Most of these applications are not compute-intensive, and are therefore not suited for running on the compute server.

Figure 56–1: DECwindows Components



56.1.2.1 The X Window System

The X Window System provides the base upon which DECwindows is built. It consists of the following components:

- DECwindows server and device drivers
- Network protocol and transport mechanism
- Xlib and Xtoolkit programming libraries

[2] Please note the difference in definition of servers and clients in the compute server environment and the DECwindows environment. In this document we will avoid such confusion by using the terms exactly as presented in Figure 56–1 when referring to the client-server relationship in DECwindows.

56.1.2.1.1 DECwindows Server and Device Drivers

The DECwindows server and its associated device drivers handle window management, drawing operations, and user input. *The Mica implementation of DECwindows does not include these components.*

56.1.2.1.2 Network Protocol and Transport Mechanism

The X network protocol specifies the common language spoken by DECwindows clients and DECwindows servers. It allows an application and its user interface to be logically separated across a network.

The transport mechanism used by Mica DECwindows is DECnet. Because there is no DECwindows server for Mica, a local transport mechanism is not needed.

Direct support for other transport mechanisms at FRS, for example TCP/IP, is not provided.

56.1.2.1.3 Xlib and Xtoolkit Programming Libraries

Xlib provides the lowest-level applications interface to the system. It acts as a thin veneer over the network protocol and transport mechanisms, converting procedure calls into packets that are transmitted to the DECwindows server. Xlib provides basic resource management and bitmap graphics services. Examples of the resources managed through Xlib are windows, color maps, and input devices.

The Xtoolkit is a library layered on Xlib. It gives applications tools for building high-level user interface objects like menus and scroll bars. The Xtoolkit is often considered to be a part of the DECtoolkit, which is described in Section 56.1.2.2.1.

56.1.2.2 Application Programming Libraries

This section describes the programming libraries that are used to develop DECwindows applications.

56.1.2.2.1 The DECtoolkit

The DECtoolkit determines the application model for DIGITAL and third party software tailored for the DECwindows environment. It establishes the conventions and styles that are encouraged for applications that share a DECwindows workstation. Applications use the DECtoolkit to build user interfaces that look and feel like integrated members of the DIGITAL computing environment.

The DECtoolkit is built as a superset of the Xtoolkit. It comprises:

- Xtoolkit intrinsics—Tools for creating, managing, and modifying user interface objects (*widgets*)
- DECwindows widgets—Common user interface abstractions such as scroll bars, menus, and buttons
- Utility routines—Functions that perform common tasks like cut and paste

56.1.2.2.2 DDIF Toolkit

The DDIF Toolkit provides routines for creating, reading, and writing Digital Document Interchange Format (DDIF) files. DDIF files provide an interchange medium for the exchange of compound text and graphics images between applications.

56.1.2.3 Implementation Strategy

DECwindows is currently being implemented for VAX/VMS, VAX/ULTRIX, PRISM ULTRIX, and MS-DOS™ . Portability is a high-priority goal for these implementations; code sharing is prevalent. Most of Mica DECwindows will be implemented by porting the VAX/VMS DECwindows V1.0 code to Mica.

The following components of DECwindows will be ported by DECwest for Mica FRS:

- Xlib
- Xtoolkit
- DECtoolkit

Additional DECwindows software, such as the DDIF Toolkit, will be ported by SDT for Mica FRS. Further details are TBS.

The Mica X transport mechanism (the interface between Xlib and DECnet) will be based on its VAX/VMS counterpart, but it will not be a port of the VAX/VMS code.

The following components of DECwindows will *not* be available on Mica at FRS:

- DECwindows server and device drivers
- User environment
- Layered libraries
- Core applications

56.1.2.4 Dependencies

The development of Mica DECwindows depends upon the availability of source code from VAX/VMS and on the availability of a PRISM C compiler. The source code will be available by August 1988.

™ MS-DOS is a trademark of Microsoft Corporation

CHAPTER 57

MISCELLANEOUS RUN-TIME LIBRARY ROUTINES

57.1 Overview

This chapter describes the following three groups of routines that are contained in the Mica applications run-time library:

- Low-level math routines implemented by SDT
- Common Multithread Architecture (CMA) routines implemented by DECwest
- Print System Model (PSM) client[1] routines implemented by DECwest

These routines provide a portion of the application program interface to Mica. Most of the routines are designed to adhere to the emerging Application Integration Architecture (AIA). As noted in the list above, the definition and development of the miscellaneous run-time library routines is the result of a cooperative effort between DECwest and SDT.

Except for the Print System Model client routines, all of the miscellaneous run-time library routines are implemented directly on Mica. The Print System Model client routines are provided via remote procedure call to an implementation of PSM on the client.

Each of these three groups of library routines is discussed in turn starting with Section 57.1.2, which describes the low-level math routines.

The Mica applications run-time library also contains other application program interface routines that complement the capabilities provided by the routines described in this chapter. These additional routines are described in Chapter 58, Application Run-Time Utility Services.

57.1.1 Goals and Requirements

The miscellaneous run-time library routines share many of the goals and requirements of the AIA program. Requirements include:

- Library routine interface implementations must be feasible on all Glacier client systems.
- Library routine definitions must allow for implementations with good performance.
- Library routine implementations must be compatible with other non-Mica implementations of the routines.

Goals include:

- To provide as complete a program interface to Mica as possible, without including nonportable concepts or constructs into the libraries.

[1] The term "client" in this context refers to a client of the PSM, namely a program running on the compute server that invokes PSM to perform some action on the *compute server client*.

- To provide a set of routines that are architected in such a fashion as to allow efficient library routine code implementations on all Glacier client systems.

Nongoals include:

- The code for these library routines must be inherently portable. The AIA architecture requires only that the *interfaces* to AIA routines be portable.
- The routines provide interfaces to every underlying operating system capability or architecture-specific hardware feature.
- The performance of these routines must on average exceed that of similar, non-AIA operating-system- or architecture-specific routines. \There is a cost for portability.\

57.1.2 Low-Level Math Routines

Math support routines exist at two levels on Mica:

- A set of low-level routines designed for use by language run-time libraries and other run-time library routines where absolute performance is paramount. The interfaces to these routines are compatible with the VAX/VMS implementations of the routines.

  The low-level math routines are described in this chapter. Table 57–1 lists the entry points for these routines.

- A set of high-level routines with AIA-conformant interfaces. These routines are used where absolute performance is secondary to portability.

  The high-level math routines are part of the Application Run-Time Utility Services (ARUS) and are described in Chapter 58, Application Run-Time Utility Services. The routine names do not conform to the Mica naming standard for reasons of compatibility with prior implementations.

Table 57–1: Low-Level Math Routines

mth$abs	*mth$cos*	*mth$gfloor*	*mth$iior*	*mth$real*
mth$acos	*mth$cosd*	*mth$gfloti*	*mth$iishft*	*mth$sgn*
mth$acosd	*mth$cosh*	*mth$gflotj*	*mth$iisign*	*mth$sign*
mth$aimag	*mth$csin*	*mth$gimag*	*mth$imax0*	*mth$sin*
mth$aimax0	*mth$csqrt*	*mth$gint*	*mth$imax1*	*mth$sincos*
mth$aimin0	*mth$cvt_d_g*	*mth$glog*	*mth$imin0*	*mth$sincosd*
mth$aint	*mth$cvt_da_ga*	*mth$glog10*	*mth$imin1*	*mth$sind*
mth$ajmax0	*mth$cvt_g_d*	*mth$glog2*	*mth$imod*	*mth$sinh*
mth$ajmin0	*mth$cvt_ga_da*	*mth$gmax1*	*mth$inint*	*mth$snglg*
mth$amax1	*mth$dim*	*mth$gmin1*	*mth$inot*	*mth$sqrt*
mth$amin1	*mth$exp*	*mth$gmod*	*mth$jiabs*	*mth$tan*
mth$amod	*mth$floati*	*mth$gnint*	*mth$jiand*	*mth$tand*
mth$anint	*mth$floatj*	*mth$gprod*	*mth$jidim*	*mth$tanh*
mth$asin	*mth$floor*	*mth$greal*	*mth$jieor*	*mth$umax*
mth$asind	*mth$gabs*	*mth$gsign*	*mth$jifix*	*mth$umin*
mth$atan	*mth$gacos*	*mth$gsin*	*mth$jigint*	*ots$divc*
mth$atan2	*mth$gacosd*	*mth$gsincos*	*mth$jignnt*	*ots$divcg*
mth$atand	*mth$gasin*	*mth$gsincosd*	*mth$jint*	*ots$mulcg*
mth$atand2	*mth$gasind*	*mth$gsind*	*mth$jior*	*ots$powcc*
mth$atanh	*mth$gatan*	*mth$gsinh*	*mth$jishft*	*ots$powcgcg*
mth$cabs	*mth$gatan2*	*mth$gsqrt*	*mth$jisign*	*ots$powcgj*
mth$ccos	*mth$gatand*	*mth$gtan*	*mth$jmax0*	*ots$powcj*
mth$cexp	*mth$gatand2*	*mth$gtand*	*mth$jmax1*	*ots$powgg*
mth$cgabs	*mth$gatanh*	*mth$gtanh*	*mth$jmin0*	*ots$powgj*
mth$cgcos	*mth$gcmplx*	*mth$iiabs*	*mth$jmin1*	*ots$powglu*
mth$cgexp	*mth$gconjg*	*mth$iiand*	*mth$jmod*	*ots$powii*
mth$cglog	*mth$gcos*	*mth$iidim*	*mth$jnint*	*ots$powjj*
mth$cgsin	*mth$gcosd*	*mth$iieor*	*mth$jnot*	*ots$powlulu*
mth$cgsqrt	*mth$gcosh*	*mth$iifix*	*mth$log*	*ots$powrj*
mth$clog	*mth$gdble*	*mth$iigint*	*mth$log10*	*ots$powrlu*
mth$cmplx	*mth$gdim*	*mth$iignnt*	*mth$log2*	*ots$powrr*
mth$conjg	*mth$gexp*	*mth$iint*	*mth$random*	

57.1.3 Common Multithread Architecture Routines

The Common Multithread Architecture (CMA) specifies the essential semantics of executing multiple threads of control within a single process's address space. CMA also defines a set of capabilities used to execute these threads on different processors of a multiprocessor system. A series of CMA routines with system-independent interfaces are included with Mica. These routines provide for the creation and control of threads, and provide a full set of synchronization and notification primitives.

CMA is described in detail in the CMA draft functional specification. Table 57–2 lists the entry points for the CMA routines. The routine names do not conform to the Mica naming standard for reasons of compatibility with the Common Multithread Architecture.

Table 57–2: CMA Routines

thd$abort_thread	*thd$enable_asynch_exception*	*thd$read_barrier*
thd$add_thread_exit_handler	*thd$end_critical*	*thd$read_event*
thd$begin_critical	*thd$enqueue_ast*	*thd$read_mutex*
thd$clear_event	*thd$enqueue_asynch_exception*	*thd$reschedule_thread*
thd$create_barrier	*thd$enter_serial_region*	*thd$set_context*
thd$create_event	*thd$get_context*	*thd$set_debug_exit_handler*
thd$create_mutex	*thd$get_cpu_time*	*thd$set_debug_init_routine*
thd$create_serial_region	*thd$get_current_thread_id*	*thd$set_pause_enable*
thd$create_thread	*thd$get_number_of_processors*	*thd$set_priority*
thd$dcl_thread_ast	*thd$get_thread_info*	*thd$thread_ast_handler*
thd$decr_barrier	*thd$incr_barrier*	*thd$timed_lock_mutex*
thd$delay_thread	*thd$leave_serial_region*	*thd$timed_wait_barrier*
thd$delete_barrier	*thd$lock_mutex*	*thd$timed_wait_event*
thd$delete_event	*thd$next_thread*	*thd$unlock_mutex*
thd$delete_mutex	*thd$notify_event*	*thd$wait_barrier*
thd$delete_serial_region	*thd$no_thread*	*thd$wait_event*
thd$dequeue_ast	*thd$pause_thread*	
thd$disable_asynch_exception	*thd$permit_asynch_exception*	

57.1.4 Print System Model Client Routines

The PSM client routines provide the program interface to the Print System Model. These routines are not yet specified; they will be specified by the PSM task group in 1988.

The PSM client routines are implemented as RPC stubs that perform RPC calls to the actual implementation of the routines on the client system.

57.1.5 Open Issues

- The concept of seamlessness between Glacier and its clients suggests that the miscellaneous run-time routines need to be implemented at or near FRS on all possible Glacier client systems. This increases the overall effort and is potentially problematic under the current manpower constraints.
- This chapter discusses the *scalar* low-level math routines only. No mention is made of the vector math routines. The interface for these routines is TBD and is expected to closely match the VAX implementation of the routines. The vector routines are not user-visible. The Mica implementations of the vector routines are not guaranteed to be bit-for-bit equivalent to their VAX/VMS counterparts.
- This chapter does not include the definition of the Basic Linear Algebra Subprogram (BLAS), a series of public domain low-level math routines. We assume that Mica will provide at least two versions of this capability: the public domain BLAS, and a DIGITAL-modified vectorized BLAS. The library the user links his or her image against determines which of these versions of BLAS a user will use.
- The PSM interface routines are not currently defined. The definition is expected in mid-1988. If not produced in time, a possible fallback is to provide RPC access to client-specific print facilities.

CHAPTER 58

APPLICATION RUN-TIME UTILITY SERVICES

58.1 Overview

This chapter describes the Mica implementation of the Application Run-Time Utility Services (ARUS) library. This library contains routines that provide the application program interface to Mica on Glacier. These routines are designed to adhere to the emerging Application Integration Architecture (AIA). The definition and development of ARUS on Mica is the result of a cooperative effort between DECwest and SDT. The major part of the implementation of ARUS is performed by SDT.

There are several discrete groups of routines contained in ARUS. Each of these groups is discussed in turn starting with Section 58.1.2.1, which describes the ARUS routines used to allocate and deallocate virtual memory.

The Mica applications run-time library also contains other application program interface routines that complement the capabilities provided by the routines described in this chapter. These additional routines are described in Chapter 57, Miscellaneous Run-Time Library Routines.

58.1.1 Goals and Requirements

ARUS shares many of the goals and requirements of the AIA program. Requirements include:

- ARUS routine interface implementations must be feasible on all Glacier client systems.
- ARUS routine definitions must allow for implementations with good performance.
- ARUS routine implementations must be compatible with other non-Mica implementations of the routines.

Goals include:

- To provide as complete a program interface as possible to contemporary DIGITAL-supplied operating systems such as Mica, VAX/VMS, and ULTRIX without including nonportable concepts or constructs.
- To provide a set of routines that are architected in such a fashion as to allow efficient library routine code implementations on all such contemporary DIGITAL operating systems.

Nongoals include:

- The code for ARUS routines must be inherently portable. (The AIA architecture requires that only the *interfaces* to AIA routines be portable.)
- ARUS routines provide interfaces to every underlying operating system capability or architecture-specific hardware feature.
- The performance of ARUS routines must on average exceed that of similar, non-AIA operating-system or architecture-specific routines. \There is a cost for portability.\

58.1.2 ARUS Routines

Although the ultimate version of ARUS will include a wide range of routines, the FRS offering is necessarily limited in scope. The FRS version of ARUS comprises those routines needed to support the FRS layered products and bundled utilities. This section discusses only the utility RTL capabilities for those areas in which there are FRS requirements.[1]

ARUS is composed of two conceptually different types of routines: generic operating system services and general purpose utility routines.

The generic operating system services provide, in an operating-system- and architecture-independent manner, those services normally associated with an operating system, such as virtual memory allocation. These routines are described starting at Section 58.1.2.1.

The general purpose utility routines provide access to common capabilities generally identified with run-time libraries, such as various data conversion routines. These routines are described starting at Section 58.1.2.7.

58.1.2.1 User Mode Virtual Memory Allocation/Deallocation Routines

ARUS contains user-level memory allocation and deallocation routines similar to the VAX/VMS LIB$VM routines. Unlike the LIB$VM routines, the ARUS routine interfaces do not use hardware-specific allocation units, such as pages. All quantities are expressed in terms of bytes.

\It is interesting to note that in a measurement made of the VMS RTL, the memory management routines were the most frequently used of any RTL routines by a factor of 10. The performance of these routines is critical, especially of *asi$get_memory*.[2]

User mode virtual memory allocation/deallocation routines include:

- *asi$get_memory*—mandatory for FRS
- *asi$free_memory*—mandatory for FRS
- *asi$create_memory_zone*—mandatory for FRS
- *asi$delete_memory_zone*—mandatory for FRS
- *asi$reset_memory_zone*

58.1.2.2 Condition Handling Routines

The ARUS condition handling routines provide an AIA-compatible interface to the Mica condition handling system. They allow the user to raise, modify, handle, and obtain information about conditions in an operating-system-independent manner.

The condition handling routines implement a dynamic condition dispatching environment whose semantics are based on the order of procedure invocation. This style of condition handling is identical to that present on VAX/VMS, Mica, and PRISM ULTRIX. The implementation of these routines utilizes the underlying operating-system-specific condition handling features. Note, however, that these routines do not operate with the traditional UNIX™ static signal handling capabilities.[3]

---

[1] The document "Overview of a New Utility RTL" by Al Simons (contained in the "AIA Strawman") contains descriptions of capabilities for the eventual ARUS library that are not represented in this chapter. All such omissions indicate that the capability described is not a realistic FRS deliverable.

[2] The spelling of all ARUS routine name prefixes, is TBD. The final routine names will have prefixes that serve to reinforce the logical grouping of the routines.

™ UNIX is a trademark of AT&T

[3] That is, the condition handling routines available in UNIX whose actions are determined by the contents of a program's "signal vector." For more information about these incompatible condition handling routines, please see Chapter 2 of the UNIX documentation.

The Mica implementation of the ARUS condition handling routines allows for access to the information in a Mica condition record in an operating-system-independent manner. The routines do not provide access to the Mica mechanism record except in a controlled way, for example, to replace the return value registers contained therein.

Note that these routines do not provide the capability of VAX/VMS routines LIB$ESTABLISH and LIB$REVERT. As discussed in Chapter 11, Condition, Exit, and AST Handling, those routines do not exist on Mica.

Condition handling routines include:

- *arus$raise_condition*—mandatory for FRS (FORTRAN, Pascal)
- *arus$replace_condition*
- *arus$add_primary_condition*
- *arus$add_secondary_condition*
- *arus$examine_condition*—mandatory for FRS (for applications not coded in Pillar)
- *arus$unwind_to_caller*—mandatory for FRS (FORTRAN, Pascal)
- *arus$unwind_to_exit*—mandatory for FRS (FORTRAN, Pascal)
- *arus$store_return_value*—mandatory for FRS (FORTRAN, Pascal)
- *arus$examine_return_value*
- *arus$add_primary_handler*—not mandatory if DEBUG goes straight to the system as expected
- *arus$add_last_chance_handler*—mandatory for FRS (FORTRAN, Pascal)
- *arus$delete_primary_handler*
- *arus$delete_last_chance_handler*

\It has not been decided whether there will be routines to map conditions from the underlying system's condition facility into common AIA conditions, or whether there will be routines to provide the means to obtain the condition name in a system-independent manner.

The question is: how does an application test for a condition such as end-of-file when the language does not provide that mapping? Will an ARUS routine map SS$_ENDOFFILE to the equivalent PRISM ULTRIX and Mica condition names or is that the responsibility of the application?

How thoroughly can we isolate the user from the underlying condition handling system?\

58.1.2.3 Date and Time Conversion Routines

The date and time conversion routines are used to convert internal format time into text, text into internal format time, and to obtain and manipulate internal format time values. They allow flexibility of natural language and format in both directions of conversion. These routines recognize and process the DIGITAL standard internal time format, as specified in standard <TBS>. On ULTRIX, there are additional routines to convert between the UNIX standard internal time format and the DIGITAL standard format.

Date and time conversion routines include:

- *aur$get_system_time*—mandatory for FRS
- *aur$format_date_time*—mandatory for FRS
- *aur$format_relative_time*—mandatory for FRS
- *aur$convert_date_string*—mandatory for FRS
- *aur$convert_relative_time_string*—mandatory for FRS

- *aur$free_date_time_context*—mandatory for FRS
- *aur$get_date_format*—mandatory for FRS
- *aur$get_maximum_date_length*—mandatory for FRS
- *aur$cvt_to_numeric_rel_time*—mandatory for FRS
- *aur$cvt_to_numeric_abs_time*—mandatory for FRS
- *aur$cvt_from_numeric_rel_time*—mandatory for FRS
- *aur$cvt_from_numeric_abs_time*—mandatory for FRS
- *aur$cvt_to_binary_rel_time*—mandatory for FRS
- *aur$cvtf_to_binary_rel_time*—mandatory for FRS
- *aur$cvt_from_binary_rel_time*—mandatory for FRS
- *aur$cvtf_from_binary_rel_time*—mandatory for FRS
- *aur$cvt_from_binary_abs_time*—mandatory for FRS
- *aur$init_date_time_context*—mandatory for FRS
- *aur$add_mixed_times*
- *aur$add_relative_times*
- *aur$subtract_absolute_times*
- *aur$subtract_relative_times*
- *aur$subtract_mixed_times*
- *aur$compare_relative_times*
- *aur$compare_absolute_times*
- *aur$get_users_language*—mandatory for FRS

58.1.2.4 String Mapping Routines

The string mapping routines provide the ability to map one string to another. The complete architecture for these routines provides for a capability similar to that available with VAX/VMS logical names, including the ability to have secure mappings.

The FRS offering of string mapping routines is more modest. At a minimum level of capability for FRS, these routines provide a uniform access to the underlying operating system string mapping capability. Such string mapping capabilities are known as logical names on VAX/VMS and environment variables on ULTRIX. This FRS support includes the ability to map a string to a single string, but without any protection from user modification of the mapping.

\We don't like these routine names. "String mapping" requires too much explanation and "logical name" is too embedded in the past, when the primary use for these mappings was providing device name independence. For this chapter, we'll use the term string mapping, but suggestions are welcome.\

String mapping routines include:

- *arus$create_string_mapping*—mandatory for FRS
- *arus$map_string*—mandatory for FRS
- *arus$delete_string_mapping*—mandatory for FRS
- *arus$create_string_mapping_table*

- *arus$delete_string_mapping_table*

58.1.2.5 Process Information Routines

Pascal has a requirement to obtain the amount of CPU time consumed by the process. That is the only currently known requirement for process information routines.

58.1.2.6 Command Language Interpreter Interface Routines

The command language interpreter (CLI) interface routines are used to provide a portable method for applications to receive and parse simple command lines. The format of the command lines is operating system specific and these routines only enforce the concepts of command verb, command parameter, command qualifier, and so on, without resorting to describing the lexical representation of these entities. The method for describing commands, parameters, and qualifiers is <TBS>.

The CLI interface routines also provide for obtaining the unparsed command line. Additionally, a routine is provided to meet the requirement of the FORTRAN RTL to be able to pause program execution and return control to the CLI.

58.1.2.7 Data Conversion Routines

Virtually all of the capabilities present in the VAX/VMS OTS$ routines are required at FRS to support FORTRAN. Please see the documented OTS$ definitions.

58.1.2.8 Text String and Message Formatting Routines

The capability needed for text string and message formatting is similar to the $FAO system service on VAX/VMS, and the *printf* statement in the C language. Like those facilities, the Mica text string and formatting routines are driven by a control string. Unlike those facilities, they include inherent support for internationalization.

Text string and message formatting routines include:

- *arus$format_text_string*—mandatory for FRS
- *arus$get_message_text*

58.1.2.9 String Routines

The string routines handle string allocation, copying, and deallocation. They closely resemble the current VAX/VMS STR$ routines that provide these capabilities. Please refer to the VAX/VMS documentation.

58.1.2.10 Table-Driven Parsing Routines

FORTRAN NAMELIST I/O currently utilizes the VAX/VMS routine named LIB$TPARSE to perform the parsing actions required. This general capability should be provided eventually in ARUS; if it is not available at FRS, the FORTRAN RTL will have to provide its own parsing routines.

58.1.2.11 Math Routines

Math support routines exist at two levels on Mica:

- A set of low-level routines designed for use by language run-time libraries and other callers where absolute performance is paramount. The interfaces to these routines are compatible with the VAX/VMS implementations of the routines. The low-level routines are described in Chapter 57, Miscellaneous Run-Time Library Routines.
- A set of high-level math routines with AIA-conformant interfaces. These routines are used where absolute performance is secondary to portability. The high-level routines are described in this chapter. Table 58–1 lists the entry points for these routines.

Table 58–1: High-Level Math Routines

math$tbs

58.1.3 Open Issues

- How to provide transportable condition handling is the area that is currently least understood. We believe that the routines described in Section 58.1.2.2 are necessary and feasible. Our current model may, however, change over the next several months as we learn more in this area.
- The most pressing issue in the area of the math routines is the lack of a definition of AIA-conformant math routine interfaces. This is delayed by the lack of a precise definition of the phrase "AIA-conformant."
- The concept of seamlessness between Glacier and its clients suggests that ARUS needs to be implemented at or near FRS on all possible Glacier client systems. This increases the overall effort and is potentially problematic under the current manpower constraints.

Glossary

GLOSSARY

access violation: An attempt to reference a virtual address to which the protection field in the PTE indicates the reference is not allowed in the specified access mode.

activated image: An image file that has been laid out in the address space of a process. All relocations and fixups have been performed, and control can be transferred to defined entry points within the image.

active partner SYSAP: The SYSAP that initiates a connection to another SYSAP.

adapter: An adapter is a communication interface that connects the XMI bus to the Computer Interconnect (CI) bus. The main function of the adapter is to move information between the Mica host and another CI node. As used here, adapter refers to a specialized port.

address space: The set of all possible virtual addresses available to a process.

address space number: A 16-bit number that is unique for each address space in the balance set. See the *PRISM System Reference Manual* for more details.

address space tables: Structures that reside in the last 4 MB of hyperspace used for managing the process's address space such as the working set list.

AIA: Application Integration Architecture.

ARUS: Application Run-Time Utility Services.

AST: See *asynchronous system trap*.

AST handler: A procedure that is intended to receive notification of a user-mode AST. These procedures are part of the program and are associated with a particular event or system service completion notification required by the thread during its execution.

AST handling facility: The Mica AST handling facility provides a mechanism for delivering asynchronous event notification in user mode to threads.

AST object: A kernel object used to interrupt the execution of a thread and cause a procedure to be called in a specified processor mode. An AST object is in the category of kernel objects called control objects.

asynchronous system trap (AST): An event that occurs asynchronous to a thread's execution, causing the thread's normal execution to be interrupted and an AST handler to be called. An AST cannot occur unless the thread has established an AST handler for it.

atomic name: A name in the module name table that is not qualified by another name.

autoloader: A routine supplied with the Mica system that performs the dynamic activation of shareable images at run time.

autoload routine: See *autoloader*.

autoload vector: An autoload vector contains the information needed by the transfer routine to dispatch to either the autoloader or the target routine. It also contains a self-relative pointer to the information needed by the autoloader to fixup the target routine's image. (See also *image autoload vector*.)

backlink: A link from a file or directory to its parent directory.

backlink path name: The "backwards" file name path, as represented by the sequence of backlinks from a given file.

balance set: The set of all address space working sets currently resident in physical memory.

balance set manager: A system thread executing in kernel mode, responsible for increasing the number of free pages in memory.

binder: The system thread of the upper-level function processor that initiated the binding operation.

binding: *(1)* The process of joining one or more lower-level FPUs to a single upper-level FPU. *(2)* A data structure containing information linking an RPC client to an RPC server.

block: The 512-byte unit of data that is transferred to and from mass storage devices.

block data transfers: Allow arbitrary quantities of data to be transported between systems. All data is guaranteed to arrive in the order sent and without duplication, or an error condition is reported to the sender.

bootable image: Bootable images are just code and data with no image header that are loaded by the console according to the *PRISM System Reference Manual*.

bound job: A Mica User, Job, Process, Thread hierarchy whose execution is controlled by compute server support software and whose context information is derived from that of the user who initiated execution of the bound job.

buffer handle: The location of the start of the buffer header.

buffer header: A data structure found at the start of each allocated recieve buffer.

built-in self tests: Logic tests built into hardware components.

bundled shadowing: Shadowing technique used by VMS that relies on the disk controller to support some of the shadow set maintenance.

callback: A call from an RPC server procedure to a procedure in the calling client.

callback table: A data structure containing the callback entry points received by a lower-level function processor from an upper-level function proessor. The callback table is created in the channel object FPU data area.

calling standard: See *PRISM calling standard*.

captured: A process in which parameters are copied to safe storage within the executive's address space (typically the IRP).

channel: An object that specifies a path or a point of connection to an FPU.

channel index: A unique identifier assigned to a channel by the NI function processor. The NI function processor assigns this channel when the *io$c_configure_channel* function is invoked. The channel index identifies the channel for packet reception.

Cheyenne: A database server that provides DDA-compliant relational database services to applications executing on a VAX front end. Cheyenne includes Mica, Quartz, and Stone.

class: For the Monitor Utility, a group of data items that provide a statistical measure of the performance of a particular subsystem.

client: The calling program in an RPC environment.

client context server: A component of compute server support software that executes on the client system. Together with the Glacier job controller server, the client context server provides the mechanism by which user Mica images are executed on Glacier.

client installation: A software installation performed on a client machine, for example, a VMS machine.

client stub: A routine in the client stub module that is called by the user code and routes the call to the server using the RPC run-time facility.

clone: Duplicate.

cluster: A cluster of program sections combined into one or several image sections. Program sections can be clustered either by specifying the programs sections directly, or by specifying the modules that contain the program sections.

cluster factor: The unit of space allocation on Files–11 volumes.

CMA: Common Multithread Architecture.

code section: A section containing all the executable code for a module. It is directly generated by the compiler and is not modified by the linker, except to combine like-named PSECT (program section) contributions into image sections.

COM: See *copy on modify*.

combined priority: See *thread priority*.

command ring: An NI controller-related data structure located in the host memory. It is used to communicate the NI function processor's special command and transmit packet requests to the NI controller.

composite object module: A module created as the result of merging multiple object modules into a single object module; when this is done, all intermodule relationships are resolved, PSECTs are concatenated, and a new symbol table is generated.

compound name: A name in the module name table that is qualified by other names. Compound names provide a means for languages to implement multiple name spaces in a way supported by both the linker and librarian.

condition: An error state that results from an error encountered during thread execution. When a condition is raised, the thread's execution is interrupted and the thread starts executing a system-supplied dispatch procedure, which locates a condition handler.

condition handler: A procedure written as a part of a program or supplied by a run-time facility to handle conditions if they occur during the execution of that program.

condition handling facility: The Mica condition handling facility provides the mechanism by which condition handlers are found and established (either at runtime or compile time). This facility provides a mechanism by which all error conditions encountered during a thread's execution may be reliably handled by the thread in a controlled manner.

condition record: A data structure that contains all condition values and arguments associated with a condition.

condition vector: Each quadword-aligned entry in a condition record. All condition-specific arguments present in a condition vector are in descriptor format.

connection: The logical link between two SCS clients. The state of the connection between two clients must be open for them to communicate. A connection is implemented on top of a virtual circuit. There are many connections to each virtual circuit. If the virtual circuit "breaks," then all connections that are implemented on that virtual circuit are also broken.

context handle: A handle that identifies state information (context) being held by a server on behalf of a client.

controller: A controller is the hardware interface between the XMI bus and a directly-connected device.

control object: A kernel object used to direct the operation of the kernel and to control processor execution. Control objects differ from dispatcher objects in that they are not used for synchronization, cannot be waited on, and do not have a state.

control space: A 64-megabyte region of virtual address space reserved for address space specific kernel structures, such as the kernel stack.

copy on modify (COM): The method used by memory management to allowing sharing of data. A page may be read freely, but it must be copied before the modification can be made.

counterpart: One of two or more logical block units in a shadow set. Each counterpart holds the same data as every other counterpart in the shadow set.

create-if-nonexistent: This option can be specified at disk file creation time. It indicates that the specified file will be created if a file of the same name does not exist in the specified directory.

credit: A "send credit" is the permission for an SCS client to send one message to a remote client. A "receive credit" is the permission given to a remote SCS client to send one message.

datagram free queue (DFREEQ): A datagram free queue is used by a Computer Interconnect (CI) adapter as a buffer source to format and deliver unsequenced CI packets. A seperate queue is established for each CI adapter by the device function processor controlling the adapter. The device function processor maintains the buffers allocated to this queue.

datagrams: Relatively short messages that have a high probability of being received by the partner. There are no guarantees that the partner will receive the datagram, that the datagram will not be duplicated, or when the datagram will be delivered. Datagrams are typically used to send error log packets.

datagrams: Short messages used by the MSCP controllers to notify the host of certain events.

data relocation table: A table describing all fixups that must be performed by the linker to the data and linkage sections of the module, based on program section addresses.

data section: A section containing all the data defined in the module. Some of this data is read only and some is read/write. This section also contains the linkage ($LINK) section and all entry descriptors for routines defined in the module.

data transfer functions: Functions which include: reading, writing, comparing, erasing, and accessing data on disks. These functions are supported by all logical block unit function processors.

DDA: DIGITAL Database Architecture.

debug symbol table: A symbol table built by a compiler containing sufficient information for the debugger to interpret user commands and display memory contents in "the current programming language."

demand zero: A page that is initialized to contain all zeros when dynamically created in memory as a result of a page fault.

device: Any piece of hardware that can be the target of a diagnostic test.

device function processor (DFP): A type of function processor used to transport commands to the device hardware. The device function processor provides the same support as the device driver provides in VMS.

device work queue object: A repository for device work queue entries that is used to communicate between a driver thread and its interrupt service routine. A device work queue object is in the category of kernel objects called control objects.

DFP: See *device function processor*.

DFREEQ: See *datagram free queue*.

diagnostic file: An executable image file containing a diagnostic program.

diagnostic pass: The execution of all selected diagnostic tests on a particular device, one time.

diagnostic run: The execution of all selected diagnostic tests on all selected devices, for the selected number of times.

diagnostic subprocess: Diagnostic programs are run as subprocesses started by the PDM server, and are known as diagnostic subprocesses.

diagnostic subtest: A set of procedures making up a diagnostic test.

diagnostic test: A set of diagnostic subtests making up a diagnostic program.

DIGITAL Storage Architecture (DSA): The DIGITAL Storage Architecture (DSA) defines the algorithms and protocols used to communicate with disks, tapes, and mass storage controllers, along with a process for managing evolution and enhancements to these algorithms and protocols.

direct message section descriptor: A pointer to a message section.

directory: A list of files or directories on a mass storage device.

directory backlink: A backlink from a directory to its parent directory.

directory entry: A filename in a directory.

directory name: Character string that represents a directory.

directory path: A list of directory names. The first element in the list is a directory in the root directory. The second element in the list is a directory in the first directory, and so on.

disk configuration functions: Functions which include bringing a unit online, initializing a stripe or shadow set, adding a counterpart to an existing shadow set, and so on. Most of the configuration functions are unique to a specific function processor.

dispatcher: A system-supplied dispatch procedure that locates and calls condition handlers. The dispatcher executes as if it had been called immediately after a condition was raised.

dispatcher object: A kernel object that is used to control and synchronize thread access to data structures and external events.

DSA: See *DIGITAL Storage Architecture*.

DSA 1: Refers to the current generation of the DIGITAL Storage Architecture.

DSA 2: Refers to the next generation of the DIGITAL Storage Architecture. DSA 2 is under development and is not well defined at this time.

DSRI: DIGITAL Standard Relational Interface—the component of DDA that specifies a mechanism used by host programs to interface to relational database systems.

DSRL: DIGITAL Standard Relational Languages—the component of DDA that specifies the DML and DDL used within programming languages to access relational database systems.

DSRP: DIGITAL Standard Relational Protocols—the component of DDA that specifies database-related intersystem protocols, used to communicate between host programs and remote database systems or between multiple database systems.

dynamic activation: Delaying the activation of an image (into memory) until it is actually referenced.

environment: A name space in which local or internal symbols are defined. Global symbols may be viewed as being in the "root" environment and therefore do not need to be qualified by an environment name.

event object: A kernel object used to record and synchronize the occurrence of an event with some action that is to be performed. An event object is in the category of kernel objects called dispatcher objects.

executable image: An image produced by the linker, with a base address of 64K (or 10000 hex) assigned to the image. Executable images must have a transfer address or the linker generates a warning at link time.

exit handlers: There are two types of exit handlers: thread and process exit handlers. Thread exit handlers are called when a thread exits. Process exit handlers are called when the last thread in a process has finished executing the last of its thread exit handlers. Exit handlers are established using a system service and kept as a list in either the TCR (for thread exit handlers) or the PCR (for process exit handlers).

exit handling facility: The Mica exit handling facility gives threads the capability of specifying and executing procedures in user mode during thread rundown. This facility allows threads and processes to perform overall clean-up actions on their environment, deallocation of system resources, or emergency actions.

expanded file specification: A fully defaulted and translated file specification.

facility-registered status: A 64-bit value which contains a facility-registered status value.

facility-registered status value: A status value unique across the entire system.

fault on execute: Indicates that a user or kernel program attempted to execute information on that page.

fault on read: Indicates that a user or kernel program attempted to read information on that page.

fault on write: Indicates that a user or kernel program attempted to write information on that page.

field replaceable unit: A piece of hardware that can easily be replaced at a customer's site. The smallest piece of a hardware subsystem that is typically replaced at a customer's site when repairs are being made.

file: A named collection of data that is organized into blocks.

file access: Defines the type of record operations that the program will perform on the file. The record operations that can be performed are: delete, get, put, truncate and update. (See also *file share options*.)

file allocation options: Options that can specify the file space allocation amount, default extension amount, and placement control.

file attributes: Characteristics of a file that are used by software, such as RMS, to specify and determine the current condition and organization of the file.

file channel: A channel to an accessed file, as represented by the presence of the file's access types in the function code access type (FCAT) table for the channel.

filename: The character string and version used to identify a file.

filename path: The combination of a filename and a directory path. The final element in the directory path is the directory in which the filename is entered.

file organization: The arrangement of data within a file.

file reference: A function-processor-specific reference to a file, which might be used to optimize access to the file.

file share options: Defines the type of record operations that the program allows other programs sharing access to the specified file to perform. (See also *file access*.)

fix-up: An action taken by the linker to alter an image so that it becomes memory-ready.

Flint: See *PRISM ULTRIX*.

flow control: Flow control inhibits a sender from sending information until the receiver has provided a buffer to hold the information. Credit accounting is used to implement flow control.

FP context area: An area reserved by a function processor following its FP parameter record in an IRP's free area. This area is allocated by the function processor if it needs to store additional context information in the IRP.

FPD: See *function processor descriptor*.

FP parameter record: Function processor parameter records hold the user I/O parameters for the request in an internal format, or the parameters of an internal request passed from one function processor to another. In addition, an FP parameter record may hold a certain amount of internal context for the request.

FPU: See *function processor unit*.

front-end installation: See *client installation*.

function code access type (FCAT) table: A table that defines the access types for all legal function codes.

function processor: A collection of kernel-mode procedures and threads that execute I/O requests.

function processor callback: The mechanism used by a lower-level function processor to communicate with an upper-level function processor.

function processor descriptor (FPD): Each function processor has a function processor descriptor (FPD) object that maintains the addresses of each global procedure in that function processor, as well as certain function-processor-specific parameters. When the function processor is needed to process an I/O request, the address of the appropriate procedure within the function processor is looked up via the FPD object.

function processor unit (FPU): A function processor accepts requests on one or more function processor units (FPUs). An FPU is an object that represents a particular resource to higher levels of software. All requests to a resource are directed to its respective FPU.

Glacier: A high-performance compute server with vector capabilities for VAX and PRISM workstations. Glacier includes Mica and Moraine, plus the compute server support software that runs on its clients. Another implementation of Glacier includes PRISM ULTRIX and Moraine. A later implementation of Glacier includes Stone in the place of Moraine.

Glacier job controller server: A component of compute server support software that executes on Mica. Together with a client context server, the Glacier job controller server provides the mechanism by which user Mica images are executed on Glacier.

global symbol: A symbol (value or location) defined in one object module, whose value is made available by the linker to other object modules.

global symbol table: A table describing symbols defined or referenced in a module. The global symbol table parallels the module name table. That is, programs must walk both tables at the same time to obtain all the attributes of an element in the global symbol table.

hardware conditions: Conditions that occur when a thread attempts some action defined as incorrect, impossible, or not yet possible by the hardware. Such action results in a hardware exception interrupting execution, which in turn causes a condition to be raised in the thread which was executing.

host area: The area of a disk reserved for host-specific information.

host transfer list (HTL): A data structure used to describe the direct I/O buffer with an array of physical addresses. When a function processor is called at its initialize I/O parameters entry point, an HTL is created by the function processor if direct I/O is to be done.

HTL: See *host transfer list*.

hyper space: An 8-megabyte region of virtual address space reserved for mapping page tables, working set, and address space specific structures.

image: A file resulting from linking several object modules together. PSECTs are gathered into image sections, and there are no unresolved external references.

image activator: The part of the system responsible for loading image files into memory and preparing them for execution.

Image autoload vector: A set of entry descriptors generated by the linker to implement the automatic loading of shareable images at run time (rather than activating all referenced images at image activation). (See also *autoload vector*.)

Image fix-up: See *fix-up*.

Image relocation tables: A relocation table within an image describing how memory locations within the data section are fixed up once the image has been activated. The linker generates relocation tables for symbols defined within the image, symbols defined in other images, and TLS region counts.

Image section: A collection of PSECTs with like protection attributes, found only in images.

Image section descriptor (ISD): Part of an image header for a section. Contains information about the image section.

Indirect message section descriptor: A pointer to a file which contains a message section and a null message section pointer.

Initial installation: A type of standard installation used when a system is being installed for the first time.

Initialization routines: Routines called by the image activator when an image is first activated.

Interface class: Function processors sharing similar access characteristics are said to belong to the same interface class. All of the function processors in a class make up a single programming interface. Examples of common interface classes are the directory structured file system class, logical block class, logical magtape class, and so on.

Internal status: A 64-bit value which contains a facility-registered status value and a 32-bit facility-defined data entity.

Interprocessor interrupt: A synchronization mechanism used by one processor to notify another processor of pending work it is to perform.

Interrupt callback: A procedure specified by a thread to synchronize with an interrupt procedure across all processors in the system.

Interrupt object: A kernel object used by a driver thread to connect an interrupt vector in the system control block (SCB) to a device interrupt service routine, or to disconnect such a vector. An interrupt object is in the category of kernel objects called control objects.

Invalid PTE: A PTE with a zero in the VALID bit.

Invocation descriptor: A quadword-aligned data structure that provides basic information about a routine. This structure is used in calls between separately compiled routines, and in interpreting the call stack that exists at any point in the execution of an image. Entry descriptors are defined by the PRISM calling standard.

Invocation descriptor-based handlers: These handlers are located from a procedure's invocation descriptor. They are used to implement a particular language's condition handling semantics. For the Pillar language, they are used to implement structured condition handling. Invocation descriptor-based handlers are established at compile time and may be called multiple times when multiple conditions are active.

I/O parameter record: A data structure containing a combination of I/O parameters, pointers to parameters, and buffer descriptors. An I/O parameter record is defined specifically for the function processor class and function code of the request in which I/O parameters are specified.

I/O request packet (IRP): An I/O request packet (IRP) is a data structure used internally by the I/O system to represent an individual request for I/O. An IRP is created by the I/O subsystem when a request for I/O is issued and remains in memory until the I/O operation completes. During the course of an I/O operation, an IRP may be passed from one function processor to another.

IOSB: See *I/O status block*.

I/O status block (IOSB): A data structure into which status information is written by the *exec$request_io* service upon successful completion of a request. The format of an I/O status block (IOSB) is the same as the IOSB in VMS, only each field is double in size.

IRP: See *I/O request packet*.

ISD: See *image section descriptor*.

kernel mode entry page: A page which is protected as user read, kernel read, kernel write, with fault on execute enabled. Kernel mode entry pages are used to dispatch to system services

kernel object: A data abstraction used to control processor execution or synchronization. There are two kinds of kernel objects: dispatcher objects and control objects. Unlike an object defined by the object architecture, a kernel object is not directly available to user software.

layered product software: A software product not part of the base system.

level: For the Monitor Utility, the current value of a data item, that is, a "snapshot."

linkage ($LINK) section: The portion of the module data section that contains pointers to data. The linkage section is generated by the compiler, and address relocations to this section are performed by the linker, using information in the address relocation table. The linkage section must not be shareable, as it contains process-private addresses.

linkage pair: A linkage pair consists of the addresses of a procedure's invocation descriptor and entry point.

loader: The part of the system responsible for loading object modules into memory, resolving external references, and preparing object modules for execution. The loader may be implemented as part of the image activator.

local procedure call: A procedure call in which the called routine is in the same address space as the calling routine.

local RPC: A special remote procedure call in which the two address spaces are on the same system.

local status: A 64-bit value which contains a local status value and the address of a message data structure.

local status value: A status value local to a particular facility.

locate mode: Technique used for record retrieval operation in which RMS returns a pointer to the record which is in the RMS I/O buffer and the record length of the retrieved record. (See also *move mode*.)

logical block interface: A set of procedure calls used to configure disks and access data on disks. These procedures support data transfer and disk configuration functions.

major priority: That part of thread priority that is controlled by the kernel.

manager thread: A system thread, created by the NI function processor, that is associated with an NI controller. This thread is responsible for the general management of the NI function processor. This includes controller configuration and reconfiguration, powerfail recovery, allocating receive buffers, and self deletion.

mapping object: Contains a pointer to the section object for a section, and the virtual address ranges of the section.

master save set: A master file containing a directory of all other save sets. (See also *save set*.)

maximize version: This option can be specified at disk file creation time. It indicates that the specified file will be created with a version number one greater than a file of the same name in the specified directory.

mechanism record: A data structure that contains information regarding the environment at the time of a condition, together with the environment of a handler when it is called.

memory management subsystem: A combination of hardware and software functions that performs the mapping of physical address space into a process's virtual address space.

memory-ready: Ready to be loaded into memory. A memory-ready image is one requiring no fix ups.

message category: A 32-bit field in the header portion of a message; identifies a subset of a message class.

message class: A 32-bit field in the header portion of a message; identifies the set of all message categories relating to a particular subject.

message FPU: See *message function processor unit*.

message free queue (MFREEQ): A message free queue is used by a CI adapter as a buffer source to format and deliver incoming sequenced messages. The device function processor controlling the CI adapter establishes a seperate queue for each adapter. Each connection over the adapter's virtual circuits allocates buffers to the queue by using the SCA flow control mechanisms.

message function processor unit: Any one of the FPUs through which threads access the message function processor.

message section: A data structure which contains message numbers and the associated message text.

message type: A quadword specifying a message class and one or more message categories within that class.

MFREEQ: See *message free queue*.

Mica: An object-based, modular operating system that supports symmetric multiprocessing and multithreaded processes. It is the base system software for Glacier and Cheyenne. Mica is written in a high-level language (Pillar) so as to be readily maintainable and extensible.

minor priority: That part of thread priority that can be explicitly modified in order to give preference within a major priority level.

modified page writer: Writes pages from the modified page list to backing store as needed.

module: A file, containing names and related information, that conforms to the described module format.

module header: The first record in a module. All information in a module can be located directly or indirectly through information in the module header.

module name table: A table containing the names of all symbols and PSECTs defined or referenced in the module. It contains both atomic and compound names. Entries in the module name table correspond one-to-one with entries in the global symbol table.

Monitor Utility: A Mica utility that displays and records information about system resource usage.

Moraine: A multiprocessor, bounded system built by DECwest. It includes one-to-four scalar/vector processor pairs, a crossbar-switching backplane, and XMI I/O; it uses CMOS II technology.

move mode: Technique used for a record retrieval in which the data records are copied from RMS I/O buffer to the program buffer.

multicast address: A predefined datalink address associated with one or more logically related stations on the NI.

mutex object: A kernel object used to control exclusive access to a resource. A mutex object is in the category of kernel objects called dispatcher objects.

Network Interconnect: See *NI*.

NI: Network Interconnect. A network interconnect can be either an Ethernet LAN or an IEEE 802.3 LAN.

NI physical address: A datalink address associated with a particular link attachment of a node on the NI.

notification messages: Messages written to a message FPU that announce the asynchronous arrival and departure of units (these units are disks for the MSCP function processor, and counterparts within the shadow set for the disk shadowing function processor). Notification messages are used to report changes in the configuration for I/O requests.

object module: The output of a compiler, a single module generated from the source language.

observation period: For the Monitor Utility, the beginning and ending times for viewing or summarizing current or previously recorded data.

ODS file identifier: The ID number assigned to a file.

on-disk context: Context stored on the disk, which is independent from the context used by the file system.

on-disk context sector: This is a sector of reserved space at the end of the host area on the disk. This context area is used by the striping and shadowing function processors to store non-file context information.

on-line installation: A software installation that occurs while the system is performing normal activity.

operating system software: The base system software (Mica).

package definition: An instance of the definition of an RPC package, written in the package definition language, Stub.

package definition block: A global record defined in the package definition module generated by the stub generator, and used by the RPC run-time facility.

package definition language: A programming language that is used to define an RPC package.

page: A set of 8192 contiguous byte locations beginning at an even 8192-byte boundary, used as the unit of memory mapping and protection.

page fault: An exception generated by a reference to a page that is not in the working set of the faulting address space.

page fault clustering: The act of reading more than one page from the disk to satisfy a page fault.

page file format: A form of an invalid PTE which refers to a page which currently resides in the paging file.

page frame number (PFN): The high-order 32 bits of the physical address of a page in physical memory.

page frame number database: A memory-resident structure containing information about each page in physical memory.

pager: A set of executive procedures which execute in kernel mode as the result of a page fault. The pager makes the page for which the fault occurred available in physical memory so that the image can continue execution. The pager and the image activator provide the operating system's memory management functions.

page table base register: See the *PRISM System Reference Manual* for more details.

page table entry (PTE): The data structure that identifies the physical location and status of a page of virtual address space. When a virtual page is in memory, the PTE contains the page frame number which maps the virtual page to a physical page. When it is not in memory, the PTE contains the information needed to locate the page.

page table pages: Lists of page table entries, these lists manage the complete address space.

paging: The action of adding and removing pages from the working set.

paging file: The file which modified pages are written to when the physical page is reused.

passive partner SYSAP: The SYSAP to which an active partner SYSAP initiates a connection.

path: At least one path exists between any two systems that can communicate. A path corresponds to the underlying physical interconnect that joins the two systems. If multiple interconnects join two systems, there can be multiple paths between them.

PB: See *processor control block*.

PDM: See *PRISM Diagnostic Monitor*.

PDM server: The portion of the PRISM Diagnostic Monitor that resides in PRISM and controls the operation of PDM-based diagnostic programs.

PDM User Interface Module: The portion of the PRISM Diagnostic Monitor that provides PDM's user interface, and resides on the client in a client-server environment.

PFN: See *page frame number*.

physical address: The address used by hardware to identify a page in physical memory.

physical address space: The set of all possible 45-bit physical addresses that can be used to refer to locations in memory space or I/O space.

piggyback AST: An AST procedure used to unmark the state change data structure. The piggyback AST executes the *io$state_change_ast_cleanup* procedure when the state change AST is delivered to the thread by the kernel. After the piggyback AST procedure completes, the kernel delivers the normal AST to the thread, and the thread's AST procedure.

Pillar: A high-level, state-of-the-art systems programming language designed for 32-bit and 64-bit DIGITAL systems. Pillar is the software development language for the Mica operating system.

port: A port is an intelligent hardware interface to a CI bus. A CI port implements parts of the SCA.

port data block: An NI controller-related data structure used during the controller initialization. The port data block gives the NI controller the location of a data structure in the host memory to be used by the controller and NI function processor to exchange data.

power-up request object: A kernel object used to request that an AST be queued when a power recovery interrupt is generated. A power-up request object is in the category of kernel objects called control objects.

power-up status object: A kernel object used to request that a specified variable's value be set to TRUE when a power recovery interrupt is generated. A power-up status object is in the category of kernel objects called control objects.

preliminary connections: Preliminary connections are connection requests by remote SYSAP partners that are not yet accepted or rejected by the local SYSAP. The connect data structures for these requests are associated with the I/O channel used by the SYSAP to enter its name in the listen directory.

primary file specification: The file specification to which defaults are applied.

primary system thread: A type of system thread used to configure a stripe set or shadow set for modification by secondary system threads.

PRISM: New computer architecture designed to be simple, flexible, expandable, and fast. All the hardware described in this glossary uses the 32-bit PRISM architecture; the architecture is designed to permit expansion to 64 bits in the future.

PRISM calling standard: The standard sequence used to call a routine. The PRISM calling standard is defined in the *PRISM Extended Calling Standard*.

PRISM Diagnostic Monitor (PDM): A controlling environment for all loadable PRISM diagnostic programs.

PRISM ULTRIX: A world-class UNIX product tailored to PRISM workstations and compute servers that supports DECwindows and AIA. PRISM ULTRIX shares some low-level components and architectures with Mica.

procedure call: The action of invoking a procedure.

process object: A kernel object that represents the address space and control information necessary for the execution of a set of thread dispatcher objects. A process object is in the category of kernel objects called control objects.

processor control block (PB): A structure that contains processor-specific information, such as a pointer to the thread object of the current thread, and the processor-specific fork queue header.

processor index: The processor index is an index into the array that catalogs the PBs in a system. This index can be used to locate the PB for a given processor.

program section: See *PSECT*.

protocol type: A 16-bit field in the Ethernet packet format for protocol demultiplexing among multiple users of a datalink.

prototype PTE: A PTE which is created during a call to the Create Section service and is used to allow complete sharing of the pages in the section.

PSECT: Program section. PSECTs describe a contiguous piece of memory. With concatenated PSECTs, all contributions for a particular PSECT are gathered contiguously in memory. If the PSECT is overlaid, all contributions for a particular PSECT begin at the same virtual address, and the module that has the largest contribution to a PSECT defines the length of the PSECT.

PSM: Print System Model.

PTE: See *page table entry*.

Quartz: High-performance relational database server software that complies with the DIGITAL Database Architecture (DDA). See also *Cheyenne*.

queue object: A repository for queue entries that is used to synchronize activity between producer and consumer threads. A queue object is in the category of kernel objects called dispatcher objects.

quick mode: A mode of operation in which diagnostic programs execute an abbreviated testing sequence.

random access: A record retrieval or storage mode. For sequentially organized files, random access to records can be done by specifying the record's position. If the sequential file has fixed length records, the records for such files can be accessed randomly by specifying the relative record number. Indexed file records can be accessed randomly by specifying either the primary or alternate key.

rate: For the Monitor Utility, the number of occurrences per second.

reader: A thread that reads from the message function processor.

readers only: The condition in which *(a)* only readers are registered on a particular message FPU, and *(b)* no messages are pending on that FPU.

read-only area: A directory structure on the system read-only disk that contains system files that are of read-only nature.

read/write area: A directory structure on the system read/write disk that contains system files that must be updated during the life of the system. It also contains replacement files for files in the read-only area.

receive buffer pool: A pool of receive buffers allocated by the NI function processor when the *io$c_ready_fpu* function is invoked. The NI function processor attaches the recieve buffers from the recieve buffer pool to the NI controller's receive ring.

receive ring: An NI controller-related data structure located in the host memory. It is used to communicate NI function processor's receive packet requests to the controller.

record: A record is a collection of related data items that is treated as a unit.

record attributes: Defines the type of record control information associated with each record.

record format: Indicates the way in which records appear physically on the recording surface of the storage medium. Record format is defined in terms of record length. The record format can be fixed length, variable length, variable length with fixed length control, stream, or undefined.

record-locking facility: A facility that prevents access to a record by more than one thread until the initiating thread releases the record.

record position: Indicates a record's physical position in a file.

registration: The process by which threads identify themselves to the messsage function processor as either readers or writers (or both).

relative record number: A positive integer that is used to indicate the position of a fixed-length record in a sequential file.

remote procedure call (RPC): A procedure call in which the called routine is in a different address space than the calling routine.

resource: A unit or volume, as represented by an FPU.

Rock: See *Stone*.

root directory: The first directory in the directory hierarchy.

RPC: See *remote procedure call*.

sample interval: For the Monitor Utility, the time interval at which systemwide performance data is to be collected and computed.

SAP: See *service access point*.

save set: A file created by the backup utility that contains other files.

SCA: See *System Communication Architecture*.

SCB: See *system control block*.

script: A command file. An ASCII file containing PDM commands.

secondary object: An I/O object, such as a file object or a port object, that is indirectly pointed to by the FPU context in a channel object.

secondary system thread: A type of system thread used to process the I/O requests that modify a stripe set or shadow set.

section: The basic unit of sharing data among processes. A section can be a disk file, a portion of a disk file, or a paging file.

segment: An object created as a side effect of a Create Section service. Every section refers to some segment object. The segment object provides the mechanism for allowing multiple users to share pages within a section.

segment 1 PTE: The first level page table which is located using bits <31:23> of the virtual address and the PTBR.

segment 2 PTE: The second level page table which is located using bits <22:13> of the virtual address and the contents previously located in the segment 1 PTE.

semaphore object: A kernel object used to control access to a resource. A semaphore acts as a gate through which a variable number of threads can pass concurrently, up to a specified limit. A semaphore object is in the category of kernel objects called dispatcher objects.

sequenced messages: Relatively short messages that are guaranteed to be delivered to the partner port. The sender is notified if they are not delivered. Sequenced messages are always delivered in the order sent, and are never duplicated.

sequential record access: A record retrieval or storage mode that starts accessing records at a designated point of the file and continues in one-after-the-other fashion through the file. That is, the records are accessed in the order in which they physically appear in the file.

server: Software that executes remote procedure calls on behalf of a client.

server stub: A routine in the server stub module that receives an RPC and makes the call to the real server procedure.

service access point (SAP): An 8-bit field in the IEEE802 format packet for protocol demultiplexing among multiple users of a datalink.

session layer: The session layer of a network protocol tower is the user's interface into the network. The user must use this layer to establish a connection to a process on another machine.

shadowing: A process which keeps duplicate data up to date on two or more disks.

shadow set: Two or more logical block units containing the same data.

shareable image: A special form of executable image that contains a global symbol table and can be input to the linker in subsequent linking operations.

shareable image space: A one half gigabyte region of virtual address space reserved for permanently mapping shareable images.

SIU: See *Software Installation Utility*.

SNAP Protocol ID: Subnetwork Access Protocol Identification field. A 40-bit field in the SNAP SAP packet format for demultiplexing among multiple users of a single SNAP SAP address.

SNAP SAP: Subnetwork Access Protocol. A reserved SAP address for protocol multiplexing and demultiplexing among multiple users of a datalink.

software audit log: A file created and appended to by the Software Installation Utility that maintains a list of operations that occured during a software installation.

software conditions: Conditions that result from an explicit use of condition handling by a thread. Software conditions may be raised at any point during thread execution. This allows applications or language run-time libraries to notify threads that some action defined as incorrect, impossible, or not yet possible was attempted by the thread. In Mica, software conditions may occur synchronously and asynchronously to thread execution.

Software Installation Utility: The utility that is used to perform standard and special software installations. The utility is invoked through system management.

software priority: See *thread priority*.

special installation: The installation procedure used to install layered product software, third-party software, and partial system updates.

spin lock: A synchronization mechanism for implementing mutual exclusion across processors in a multiprocessor configuration.

standard installation: The installation procedure used to install the operating system software.

status value: A 32-bit value used to return information regarding the success or failure of a process, thread, I/O service, or procedure back to the thread which created or called it. There are two types of status values: facility-registered status values and local status values.

Stone: A multiprocessor high-reliability system built by DECwest. It includes one-to-four scalar/vector or scalar/scalar processor pairs, a crossbar-switching backplane, and XMI I/O; it uses CMOS III technology.

stripe fragments: A portion of a stripe residing on a single unit. All of the stripe fragments in a stripe cover the same range of logical block numbers on each unit.

stripes: Individual units of data in a stripe set.

stripe set: The virtual representation of all of the disk units used in the striping process. Higher levels of software view the stripe set as a single continuous vector of logical blocks.

striping: A process which presents two or more disks as a single disk to higher levels of software.

Stub: The Mica RPC package definition language.

stub modules: Modules containing client stubs or server stubs. Stub modules are generated by the stub generator.

Subnetwork Access Protocol ID: See *SNAP Protocol ID*.

subsection: Structure which is contained within the segment control area which provides the information to translate a virtual address to a range of virtual block numbers within a mapped file.

subsection format: Format of a prototype PTE when it refers to a subsection.

supersede version: This option can be specified at disk file creation time. A file created with this option supersedes any file in the specified directory with the same file name, file type, and version number.

SVA: See *system virtual address*.

synonym filename path: The filename path given to a file when it is entered into more than one directory, using one or more file names.

SYSAP: Functions within the operating systems of hosts, as well as the firmware of disks and tape controllers that need to communicate over the CI interconnect. Examples of SYSAPs include disk and tape class drivers, DECnet software, and the VAXcluster connection manager.

system catchall handler: A user-executable procedure, mapped in system space, that catches all improperly handled conditions.

System Communication Architecture (SCA): Defines how data traffic is handled among systems over the CI interconnect.

system control block (SCB): An architecturally defined structure that contains addresses of exception and interrupt service routines. The routines cataloged in the SCB are used to handle system interrupts and exception conditions.

system image: System images are both executable and shareable images that are loaded into system memory. These images are used by all kernel mode software.

system space: A one and one half gigabyte region of virtual address space reserved for mapping the operating system and operating system data structures.

system virtual address (SVA): A virtual address identifying a location in system space.

target system: The hardware system on which the software is being installed.

temporary marked for delete: This option can be specified at disk file creation time. A file created with this option is created without any directory entry. The file is automatically deleted when the file is closed.

temporary read/write area: A directory structure on the system read/write disk that is created by the Software Installation Utility during a special installation. Products are first placed in this directory structure before being placed in the real system read/write area.

third-party software: Software products created by vendors other than DIGITAL.

thread-local storage: See *TLS*.

thread object: The agent that executes program code and is dispatched for execution by the kernel. A thread object is in the category of kernel objects called dispatcher objects.

thread priority (or combined priority, or software priority): The importance level of a thread, in the range from 0 to 63, used by the thread dispatcher. Thread priority is divided into a major priority and a minor priority.

timer object: A kernel object used to synchronize thread activities based on the passage of time. A timer object is in the category of kernel objects called dispatcher objects.

TLS: Thread-local storage. TLS is per-thread storage with FORTRAN COMMON semantics, and storage allocated at run time.

TMSCP: Tape Mass Storage Control Protocol is the name of the interface used to communicate with DSA magnetic tape drives and controllers.

transfer address: The address of an invocation descriptor in an executable image that is called when the image is run.

transfer code: The code generated by the linker that transfers a call to a routine in another image to the autoloader.

transfer vector: The offset from the beginning of a mapped shareable image to the invocation descriptor of the routine it represents.

transition page: A page currently on the standby list or modify list, or a page in the process of being read.

translation buffer: An internal processor cache virtual to physical translations for recently used virtual addresses.

translation not valid fault: This fault invokes the pager. Also referred to as a page fault

unwind facility: The Mica unwind facility centrally provides the capability to perform nonlocal GOTOs within a thread. It is implemented as a user-mode procedure, mapped in system space, and reached via a procedure variable in the process control region.

update installation: See *upgrade installation*.

upgrade installation: A type of standard installation used when a system is being upgraded from one version to another version. During this type of installation some information from the previous version must be transfered to the new version, for example, the system authorization file.

USE: See *User-Level System Exerciser*.

User-Level System Exerciser (USE): The User-Level System Exerciser is similar to the VAX/VMS UETP. It tests device connections to the hardware and simulates load testing.

user space: A two gigabyte region of virtual address space reserved for user mode images.

VAX port queue object: A repository for VAX port queue entries that is used to communicate between a PRISM processor and a VAX port device controller. A VAX port queue object is in the category of kernel objects called control objects.

VBN: See *virtual block number*.

vectored handlers: There are two types of vectored handlers: primary and last chance. Primary vectored handlers are the first searched for when a condition is raised. The list of primary handlers is called in FIFO order with respect to when they were established. Last chance vectored handlers are called in LIFO order with respect to when they were established. Vectored handlers may only be established at runtime, by using a system service.

version: A value that distinguishes files with the same character string.

viewing interval: For the Monitor Utility, the time interval at which current or previously recorded data is to be displayed to the user screen.

virtual address: A 32-bit unsigned integer that specifies a byte location within the virtual address space.

virtual address space: The set of all possible virtual addresses that an image can reference.

virtual block number (VBN): The file-relative address of a block on a mass storage device. VBNs are 512-byte entities on the disk. If the size of the virtual blocks of the on-disk structure changes, software must convert 512-byte VBN numbers to the new values, which should be a multiple of 512 bytes.

virtual circuit: A virtual circuit is the logical link between two CI ports. The SCS layers of different systems can only communicate when a virtual circuit is open. An open virtual circuit provides sequence message service, datagram service, and block data transfer service.

virtual function processor: A type of function processor used to implement the virtual layers of the I/O system. Virtual function processors are used to implement all of the virtual-level operations, such as the file system, disk striping, virtual terminal support, and so on.

virtual memory: The set of storage locations in physical memory and on disk that is referred to by virtual addresses.

virtual page number (VPN): Bits <31:13> of the virtual address.

volume: A mass storage medium, such as a disk pack or reel of magnetic tape.

volume channel: An accessed channel to a volume FPU.

volume FPU: An FPU representing a single disk volume or a volume set.

volume set: A collection of volumes that is interpreted as a single volume by higher layers of software.

VPN: See *virtual page number*.

worker's threads: System threads created by the NI function processor. Worker threads process the receive packets and deliver the packets to the upper-layer function processor; handle the completion of packets transmission and controller command completion; queue error requests when errors occur.

working set list: The set of pages to which an executing thread can refer without incurring a page fault.

working set list entries (WSLE): The elements used to manage the working set. Each page in the working set is represented by a working set list entry.

writer: A thread that writes to the message function processor.

writers only: The condition in which only writing threads are registered on a particular message FPU.

WSLE: See *working set list entry*.