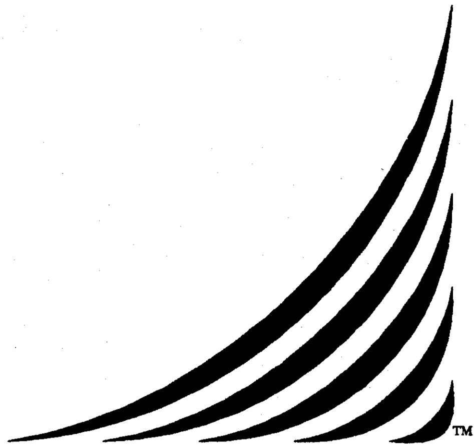
V12

™

MAINSAIL®

STREAMS and MAINKERMIT User's Guides



MAINSAIL® STREAMS and MAINKERMIT

User's Guides

24 March 1989

xidak™

Copyright (c) 1986, 1987, 1988, 1989, by XIDAK, Inc., Menlo Park, California.

The software described herein is the property of XIDAK, Inc., with all rights reserved, and is a confidential trade secret of XIDAK. The software described herein may be used only under license from XIDAK.

MAINSAIL is a registered trademark of XIDAK, Inc. MAINDEBUG, MAINEDIT, MAINMEDIA, MAINPM, Structure Blaster, TDB, and SQL/T are trademarks of XIDAK, Inc.

CONCENTRIX is a trademark of Alliant Computer Systems Corporation.

Amdahl, Universal Time-Sharing System, and UTS are trademarks of Amdahl Corporation.

Aegis, Apollo, DOMAIN, GMR, and GPR are trademarks of Apollo Computer Inc.

UNIX and UNIX System V are trademarks of AT&T.

DASHER, DG/UX, ECLIPSE, ECLIPSE MV/4000, ECLIPSE MV/8000, ECLIPSE MV/10000, and ECLIPSE MV/20000 are trademarks of Data General Corporation.

DEC, PDP, TOPS-10, TOPS-20, VAX-11, VAX, MicroVAX, MicroVMS, ULTRIX-32, and VAX/VMS are trademarks of Digital Equipment Corporation.

EMBOS and ELXSI System 6400 are trademarks of ELXSI, Inc.

The KERMIT File Transfer Protocol was named after the star of THE MUPPET SHOW television series. The name is used by permission of Henson Associates, Inc.

HP-UX and Vectra are trademarks of Hewlett-Packard Company.

Intel is a trademark of Intel Corporation.

CLIPPER, CLIX, Intergraph, InterPro 32, and InterPro 32C are trademarks of Intergraph Corporation.

System/370, VM/SP CMS, and CMS are trademarks of International Business Machines Corporation.

MC68000, M68000, MC68020, and MC68881 are trademarks of Motorola Semiconductor Products Inc.

ROS and Ridge 32 are trademarks of Ridge Computers.

SPARC, Sun Microsystems, Sun Workstation, and the combination of Sun with a numeric suffix are trademarks of Sun Microsystems, Inc.

WIN/TCP is a trademark of The Wollongong Group, Inc.

WY-50, WY-60, WY-75, and WY-100 are trademarks of Wyse Technology.

Some XIDAK documentation is published in the typefaces "Times" and "Helvetica", used by permission of Apple Computer, Inc., under its license with the Allied Corporation. Helvetica and Times are trademarks of the Allied Corporation, valid under applicable law.

The use herein of any of the above trademarks does not create any right, title, or interest in or to the trademarks.

Table of Contents

1. MAINSAIL STREAMS and MAINKERMIT User's Guides 1
  1.1. Conventions Used in This Document 1
  1.1.1. User Interaction 1
  1.1.2. Syntax Descriptions 1
  1.1.3. Temporary Features 2

I. MAINSAIL(R) STREAMS User's Guide 3

2. Overview 4
  2.1. Version 4
  2.2. Terminology 4
  2.3. Stream Features 6
  2.3.1. Summary of Stream Support and Implementation 6
  2.3.2. The Basic TTY Stream 6
  2.3.3. Reading Interrupt Characters from the TTY 7
  2.3.4. Opening Serial TTY Lines 8
  2.3.5. Setting the Baud Rate on TTY Streams 8
  2.3.6. Writing a BREAK on TTY Streams 8
  2.3.7. Clearing Pending I/O 8
  2.3.8. Scheduling of TTY Streams 8
  2.3.9. Catching Keyboard Interrupts 9
  2.3.10. Server/Client Communication 9
  2.3.11. Process Rendezvous 9
  2.3.12. Child Process Creation 9
  2.3.13. The System Shell as a Child Process 10
  2.3.14. Scheduling of Advanced STREAMS 10
  2.3.15. Timeouts 11
  2.4. Stream Types 11
  2.5. Automatic Scheduling of Stream I/O 12
  2.6. Loose Coupling of Streams to MAINSAIL 12

3. Opening and Closing Streams 15
  3.1. Making STREAMS Available to a MAINSAIL Program: the STRHDR Intmod and $initializeStreams 15
  3.2. Automatically Opened Streams: $tty and $parent 15
  3.3. The Class $stream 16
  3.4. Opening and Closing Streams: $openStream and $closeStream 18

4. Remote Modules and Remote Procedure Calls (RPC) 21
  4.1. Overview 21

4.2. Remote Module Procedures 24
4.3. Remote Module Interfaces 26
4.4. Buffer Passing Conventions 27
4.5. Example of Remote Modules between a Parent and Child Process 29
4.6. Example Clients and Servers Using Remote Modules 32
4.6.1. Use of Version Numbers 33
4.6.2. Writing Adaptable Clients 36
4.6.3. The Server Log File 37
4.6.4. Terminating a Global Server 37
4.7. Generic RPC Server: the RPCSRV Module 38
4.8. RPC Implementation Restrictions 38
4.9. RPC Efficiency Considerations 39
4.10. Parallel Processing Using RPC Calls 39
4.11. Remote Exceptions 39
4.12. C RPC 40
4.12.1. Data Type Rules 41
4.12.2. connserver and the _init Procedure 45
4.12.3. Calling Remote Procedures 45
4.12.4. The _final Procedure and close 46
4.12.5. dispose_array 46
4.12.6. Handling Exceptions in the Remote Module 46
4.12.7. Sample C RPC Session 47

5. RPC Files 50
5.1. $openRpcFile 50
5.2. $rpcFileModuleCls 51

6. Interprocess Communication: Socket Streams 56
6.1. Meaning of $eos on Socket Streams 56
6.2. Socket-Specific Fields of $stream 56
6.3. Servers and Clients: Network Protocol Modules and the SERVICE Stream Prefix 58
6.3.1. Terminology and Conventions 58
6.3.2. Service Protocol Table 59
6.3.3. The Client End 59
6.3.4. The Server End 60
6.3.5. The Service Protocol Table 64
6.4. Child Process Creation: SOCPRO and PTYPRO 65
6.4.1. Starting a Child Process Using SOCPRO 65
6.4.2. Starting a Child Process Using PTYPRO 66
6.4.3. Starting a MAINSAIL Child Process 67
6.4.4. Establishing an Additional Control Stream to a Cooperating Child 68
6.5. Terminating a Child Process 68
6.6. Process Control 69
6.7. Examples 69
6.7.1. Sprouting a Print Job and Waiting for It to Exit 69
6.7.2. Sprouting an Interactive Child 70

6.8. Process Rendezvous Communication [Not Implemented] 73
6.8.1. Creating a Rendezvous Stream Name 73
6.8.2. Symmetric Communication Using a Rendezvous Name 73
6.8.3. Server/Client Communication Using a Rendezvous Name 74
6.8.4. Underlying Implementation 74

7. Multitasking and the Scheduler 75
7.1. Cautions About Shared Data Access 76
7.2. Scheduling Coroutines for Stream I/O: $queueCoroutine 77
7.3. Voluntary Rescheduling: $msTimeout and $reschedule 77
7.4. Waiting for Descendant Coroutines: $waitForDescendants 78
7.5. Semaphores (Locks) 78
7.6. Scheduled Coroutine Map 80
7.7. Example Multitasking Programs 80
7.7.1. A Simple Terminal Emulator 80
7.7.2. Parallel Processing Using RPC Calls 81

8. Handling Keyboard Interrupts 87
8.1. Enabling and Handling Interrupts on $tty 87
8.2. Stacking of $enableInterrupts 88
8.3. Conflict between $enableInterrupt and the $noInterrupt Bit 89
8.4. Sample Program That Catches Interrupts 90
8.5. Causing Keyboard Interrupts to Occur in a Child Process 91

9. Low-level Stream I/O Procedures 92
9.1. Overview of the Low-Level Stream I/O Procedures 92
9.1.1. Timeouts 92
9.1.2. Success and Failure of I/O Operations 93
9.1.3. The General Error Return ($error) 94
9.1.4. The End-of-Stream Return ($eos) 94
9.1.5. Timeout Return ($timedOut) 95
9.2. Octets 95
9.3. Input: $readStream 95
9.4. Output: $writeStream 98
9.5. Single-Character I/O: $cReadStream and $cWriteStream 100
9.6. Miscellaneous Operations: $flushStream and $clearStream 101
9.7. Constants and Macros for Stream I/O 103

10. TTY Streams: TTYSTR 104
10.1. $tty 104
10.2. Meaning of $eos on TTY Streams 105
10.3. TTY-Specific I/O Bits 105
10.4. TTY-Specific Procedures 108
10.5. TTY-Specific Fields of $stream 109
10.6. Is My $tty Interactive? 110
10.7. Am I a Cooperative Child? 110
10.8. Half-Duplex TTY Streams 111

10.9. PTY Streams 111
10.9.1. Meaning of $eos on PTY Streams 112
10.9.2. PTY-Specific Procedures 113
10.9.3. PTY-Specific Fields of $stream 113

11. MEMORY Streams: MEMSTR 114

II. MAINKERMIT User's Guide 135

12. Overview of MAINKERMIT 136
12.1. Version 136
12.2. File Names and Types 138
12.3. File Transfer 138
12.4. Operation 138

13. MAINKERMIT Commands 140
13.1. The "SEND" Command 141
13.2. The "RECEIVE" Command 141
13.3. The "GET" Command 141
13.4. The "TEXT", "PTEXT", and "DATA" Commands 142
13.5. The "SERVER" Command 142
13.6. The "RTEXT", "RPTEXT", and "RDATA" Commands 142
13.7. The "REMOTE", "FINISH", and "REXECUTE" Commands 143
13.8. Local File Manipulation Commands 144
13.9. The "SET" Command 144
13.9.1. "SET LINE deviceName" 144
13.9.2. "SET BAUD [300|1200|2400|4800|9600]" 145
13.9.3. "SET FILETYPE [TEXT|PTEXT|DATA]" 145
13.9.4. "SET TARGET osName" 145
13.9.5. "SET PROMPT s" 145
13.9.6. "SET DEBUG" 146
13.10. The "TAKE" Command 146
13.11. The "CONNECT" Command 146
13.12. The "HISTORY" Command 146

14. Program Interface 148

Appendices

A. Server Installation Instructions 115
A.1. The Format of the XIDAK Service Protocol Table 115
A.1.1. "MYHOST hostName" 115
A.1.2. "HOSTNAME officialName alias1 alias2 ... aliasn" 116
A.1.3. "HOSTPROTOCOL hostName protocol1 protocol2 ... protocoln" 116
A.1.4. "SERVICE serviceName hostName/protocol ..." 117
A.1.5. "DEFAULTPROTOCOL protocol1 protocol2 ... protocoln" 117
A.2. Host-Dependent Service Tables 117
A.3. Installing the "gensrv" Server and the Service Protocol Table 117
A.3.1. Create a XIDAK Service Protocol Table 118
A.3.2. Add "ENTER" Subcommands to the MAINSAIL Site-Specific Startup Command File ("site.cmd") 118
A.3.3. Modify the Operating System Services Table 119
A.3.4. Arrange for "gensrv" to be Started as a Background Process 119

B. Available Network Protocol Modules 121

C. System-Specific Support For Streams 122

D. Extended C RPC Client Example 124

E. Remote Streams: NETSTR 128
E.1. Opening a Remote TTY Device 128
E.2. Explicit Gateway Mechanism 128
E.3. Underlying Implementation 129

F. A Guide to Remote File Access Using NET 130
F.1. Use of the NET Device Module 130
F.2. Execution of Modules Stored Remotely 131
F.3. Syntactic Sugar for Remote File Access 131
F.4. Known Restrictions and Limitations 131

G. XIDAK STREAMS Applications and Utilities 133
G.1. Technology Database Management System: TDB 133
G.2. Network File Access: NET Device Module 133
G.3. Bug Tracking System: UCRSYS 133
G.4. Network Server Status: SRVINF 133
G.5. MAINSAIL Kermit 134

H. Status of MAINKERMIT Version 2.0 149

I. XIDAK Operating System Abreviations Used with the "SET TARGET" Command 150

List of Examples

1.1.1-1. How User Input Is Distinguished 1
1.1.2-1. Syntax of a Mailing Address. 1
2.6-1. Using the NTTY Device Module 13
3.1-2. Use of $initializeStreams 16
4.4-1. Example Uses, Produces, and Modifies Buffer Parameters 28
4.5-1. Example Remote Module 30
4.5-3. Child Providing the FOOMOD Remote Module. 30
4.5-2. Parent Executing Remote Module FOOMOD in a Child 31
4.6-1. Example Remote Module 33
4.6-2. Example Client of the FOOMOD Remote Module Service 34
4.6-3. Example Server Providing the FOOMOD Remote Module 35
4.6.1-1. A Remote Module That Sets Its Version Numbers 35
4.6.3-1. Example "server.log" Log File 37
4.12.6-1. Use of rpc_register_jump and rpc_clear_jump 47
4.12.7-1. Compilation of Remote Module with "RPC C" Compile Subcommand . . 48
4.12.7-2. C RPC Client in "foo.c" 49
6.7.1-1. Sprouting a Print Job and Waiting 69
6.7.2-1. Sprouting an Interactive Child 70
7.7.1-2. A Simple Terminal Emulator Using STREAMS 82
7.7.2-1. Multitasking RPC Calls 84
8.1-2. Handling Keyboard Interrupts 89
8.4-1. Catching Interrupts Asynchronously 90
A.1-1. Typical Service Protocol Table 116
A.3.1-1. Minimal General Service Protocol Table between Hosts A and B 118
A.3.2-1. "ENTER" in "site.cmd" for Using Servers 118
A.3.3-1. Declaring the Service "gensrv" to BSD UNIX 119
A.3.4-1. Starting GENSRV Automatically Under BSD UNIX 120
D-1. MAINSAIL RPC Server Interface Procedure Header 124
D-2. C Client That Calls MAINSAIL Remote Module 124

List of Figures

2.3.1-1. Summary of Stream Support. 7
4.1-1. Schematic Diagram of a Remote Procedure Call 23
5.1-1. $openRpcFile 50
5.2-2. RPC File openBits 52
5.2-1. CLASS $rpcFileModuleCls. 53
7.7.1-1. Schematic of a Terminal Emulator Program 81
14-1. MAINKERMIT Module Declartion 148

List of Tables

3.1-1. $initializeStreams 15
3.3-1. User-Visible Fields of $stream and STREAMS Macros 17
3.4-1. $openStream (Generic) and $closeStream 18
3.4-2. Error Handling in $openStream 20
4.2-1. Remote Module Procedures 24
4.6.1-2. User-Visible Fields of $remoteModuleCls 36
4.12-1. C Client Arguments and MAINSAIL Server Parameters 42
4.12.1-1. ARRAY Struct Fields and Meanings 44
4.12.7-3. Compiling the C Client on a Typical UNIX System 48
6.2-1. Socket-Specific Fields of the Stream Record 57
6.3.3-1. Client Access to a Service 59
6.3.4-1. Server Procedures 60
6.3.4-2. Valid hostAndServiceName values ($getProtocols) 62
6.3.4-3. Fields of $port 63
6.4.3-1. $executableBootName 67
6.8.1-1. $createRendezvousName 73
7.2-1. Creating and Scheduling Coroutines 77
7.3-1. Voluntary Rescheduling Calls 77
7.4-1. Coroutine Synchronizing Calls: $waitForDescendants (Generic) 78
7.5-1. Semaphore Scheduler Calls 78
7.6-1. Scheduled Coroutine Map 80
8.1-1. Interrupt Catching Procedures 87
9.1.1-1. Special Timeout Values 93
9.1.2-1. Error Testing Macros 93
9.3-1. $readStream (Generic) 95
9.3-2. Buffering Modes (X = valid, - = invalid, D = Default) 97
9.4-1. $writeStream (Generic) 98
9.5-1. $cReadStream and $cWriteStream 100
9.6-1. $flushStream and $clearStream 101
10.4-1. TTY-Specific Procedures 108
10.5-1. TTY-specific Fields of $stream 109
10.9-1. Interaction Between PTY and $tty 111
10.9.2-1. PTY-Specific Procedures 113
B-1. Protocol Modules and Their Characteristics 121
C-1. Features Supported by STREAMS Implementations 122
12-1. MAINKERMIT At a Glance 137
12.3-1. Information Displayed During File Transfer 138
13-1. MAINKERMIT Command list 140
13.5-1. Server Commands Supported by MAINKERMIT 143
13.7-1. Remote Commands 143
13.8-1. MAINKERMIT Local File Commands 144
13.9-1. MAINKERMIT "SET" Options 145

13.11-1. MAINKERMIT Emulator Commands 146
I-1. XIDAK Operating System Abreviations. 150

1. MAINSAIL STREAMS and MAINKERMIT User's Guides

This document describes MAINSAIL STREAMS, a package for portable distributed applications, and MAINKERMIT, a file transfer program using the KERMIT protocol, which is implemented using STREAMS.

1.1. Conventions Used in This Document

1.1.1. User Interaction

Throughout the examples in this document, characters typed by the user are underlined. "<eol>" symbolizes the end-of-line key on a terminal keyboard; this key is marked "RETURN" or "ENTER" on most keyboards. In Example 1.1.1-1, "Prompt:" is written by the computer; the user types "response" and then presses the end-of-line key.

```
Prompt: response<eol>
```

Example 1.1.1-1. How User Input Is Distinguished

1.1.2. Syntax Descriptions

Specifications of syntax often contain descriptions enclosed in angle brackets ("<" and ">"). Such descriptions are not typed literally, but are replaced with instances of the things they describe. For example, a specification of the syntax of the address on an envelope might appear as in Example 1.1.2-1.

```
<name of addressee>
<street number> <street name>
<town or city name>, <state abbreviation>  <zip code>
```

Example 1.1.2-1. Syntax of a Mailing Address

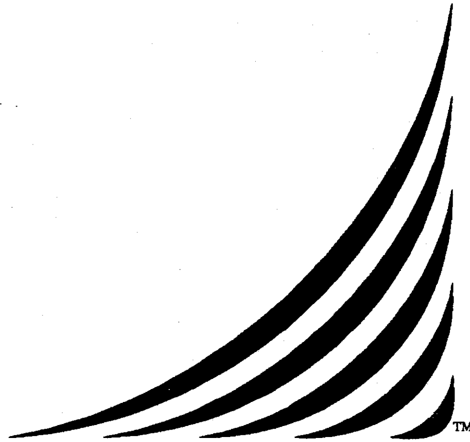
Optional elements in command or syntax descriptions are often enclosed in curly brackets ("{" and "}"). For example, a string of characters specified as "{A}B{C}" could have any one of the forms "B", "BC", "AB", and "ABC". Alternatives may be enclosed in square brackets (or curly brackets, if all alternatives are optional) and separated by vertical bars ("|"); "[A|B|C]" means "A", "B", or "C"; "{A|B}" means "A", "B", or nothing.

1.1.3. Temporary Features

Temporary features that have not acquired a final form are marked as follows:

```
TEMPORARY FEATURE:  SUBJECT TO CHANGE
```

Temporary features are subject to change or removal without notice. Programmers who make use of temporary features must be prepared to modify their code to accommodate the changes in them on each release of MAINSAIL. It is recommended that code that makes use of temporary features be as isolated from normal code as possible and thoroughly documented.



MAINSAIL® STREAMS

User's Guide

[Preliminary Documentation: 24 March 1989]

2. Overview

This document describes STREAMS, a MAINSAIL package supporting implementation of portable distributed MAINSAIL applications.

The STREAMS package is not part of the MAINSAIL runtime system, but is a separately licensable package. Due to the difficulty of implementing portable interprocess communication and process control, XIDAK cannot guarantee that that the STREAMS package will be available on all platforms that support MAINSAIL, nor can XIDAK implement all STREAMS capabilities on all plaforms for which XIDAK supports the STREAMS package. XIDAK will attempt to supply STREAMS on all platforms for which it makes technical and business sense, and to support all STREAMS capabilities on each such platform for which such capabilities can be effectively implemented. Accordingly, programs that make use of STREAMS must be written to adapt to the capabilities provided on each platform if they are to be used portably on more than one platform, and the programmer must understand that programs with absolute requirements for certain capabilites may not work at all on certain platforms.

2.1. Version

This version of the "MAINSAIL STREAMS User's Guide" is current as of Version 12.10 of MAINSAIL. It supersedes the document entitled "STREAMS, a MAINSAIL Package for Distributed Computing".

As of the date of this document, STREAMS has not yet been cast in final form. The current release of STREAMS should be considered an alpha release, subject to subsequent change. This is so that qualified customers can provide feedback on possible modifications. Accordingly, if you use STREAMS, please don't hesitate to contact XIDAK with suggested improvements or description of your problems. XIDAK does not envision any fundamental changes in the architecture of STREAMS, but does reserve the right to make changes it deems necessary to create a truly usable product (these changes are likely to include the provision of more convenient, higher-level facilities for some operations).

2.2. Terminology

"Distributed computing" refers to independent sequential algorithms that run simultaneously or asynchronously on one or more computers ("nodes"), and the intercommunication facility (STREAMS) by which the algorithms communicate.

A "process" is an operating-system-specific unit of execution with its own context (e.g., address space, primary input and output, global variables, set of open files, and other resources). For example, all current implementations of MAINSAIL consist of a single process that is started with the MAINSAIL bootstrap. Most platforms allow multiple processes to run independently of each other, without requiring mutual awareness among separate processes (except where the algorithm desires to use interprocess communication).

"Scheduled coroutines" are independent threads of execution that run within a single MAINSAIL process. Using scheduled coroutines, an application can implement cooperative multitasking within a single MAINSAIL process. Scheduled coroutines are implemented as ordinary MAINSAIL coroutines.

Scheduled coroutines are more efficient to create and schedule than processes. They can be thought of as "featherweight" processes that share the resources of a single process, including the address space, primary input and output, global variables, and open files.

Because they share data structures and resources, scheduled coroutines can communicate among themselves much more efficiently than can separate processes. Unlike programs that run as independent processes, however, procedures that are run as scheduled coroutines generally must be written with some awareness of other scheduled coroutines that may be running and sharing the data structures and resources in the same process.

When the underlying operating system supports the necessary capabilities, the STREAMS package provides these facilities related to interprocess communication and distributed computing:

- Streams: two-way channels used for communication among processes and with human users through various communications devices.

- Child process support: the creation of, communication with, and deletion of child processes controlled by a parent process. The children need not run on the same node as the parent process.

- Server and client processes: the ability of a process (the server) to establish a service with a global, unique name, and the ability of a client process to establish communication with that service, given the service's name. A single server may provide more than one service. Server and client processes may run on different nodes.

- Process rendezvous: the ability of two processes to establish communication with each other based on a rendezvous name supplied by an arbitrary third process. The two processes may run on different nodes.

- Scheduled coroutines: cooperating "featherweight" tasks running within a single MAINSAIL process that perform independent I/O functions in parallel.

2.3. Stream Features

2.3.1. Summary of Stream Support and Implementation

Figure 2.3.1-1 is a Venn diagram showing the possible support combinations for various stream-related features. Basic STREAMS, guaranteed for all STREAMS implementations, includes only the basic TTY (terminal) stream, which may be limited to half-duplex line-at-a-time operation. Various additional features of the TTY stream are available on some platforms (see below).

On some platforms, advanced STREAMS provides the ability to control processes and communicate with other processes using server/client communication. If advanced STREAMS is supported, the ability to schedule advanced STREAMS may also be supported.

Appendix C summarizes the current status of support and implementation of STREAMS on the current MAINSAIL platforms. Programs that must run on systems where only some STREAMS features are supported must use the runtime tests provided to determine which features are available. They must then adapt to the environment as best they can.

2.3.2. The Basic TTY Stream

The minimal STREAMS implementation is a single TTY stream that connects to the controlling terminal device. This stream need not be capable of independent input and output scheduling, polling, or timed reads. The STREAMS system variable $tty, declared as:

```
POINTER($stream) $tty;
```

is automatically opened to the controlling terminal.

The TTY stream may be half-duplex, in which case I/O may be restricted to a line at a time. This minimal implementation is sufficient to write, e.g., a KERMIT server, but not a KERMIT client. The minimal STREAMS capabilities provided for half-duplex TTY streams is no more than what the MAINSAIL language currently guarantees for terminal I/O on half-duplex systems.

The minimal STREAMS capabilities for full-duplex TTY streams are more than what the MAINSAIL promises, but similar to what the full-duplex MAINSAIL display modules assume.

```
+-----------------+
| BASIC STREAMS   |
|                 |
|       +-------------------------------------+
|       |        | TTY Stream Features Available on  |
|       |        | Some Systems:                     |
|       |        | Full-duplex TTY                   |
|       | Basic  | Reading interrupt characters      |
|       | TTY    | Open serial lines, set baud rate  |
|       | Stream | Write a BREAK, clear pending I/O  |
|       |        | Scheduled TTY streams, timeouts   |
|       |        | Keyboard interrupts               |
+-------+--------+-----------------------------------+
| ADVANCED        |
| STREAMS         |
|       +-----------------------------------+
|       |        | Advanced STREAMS Features  |
|       | Server | Available on Some Systems: |
|       | Client | Scheduled advanced STREAMS |
|       +--------+ Timeouts                   |
|       | Rendez-| PTY access to child process|
|       | vous   | System shell as a child    |
|       +--------|                            |
|       | Parent |                            |
|       | Child  |                            |
+-------+--------+----------------------------+
```

Figure 2.3.1-1. Summary of Stream Support

Thus, full-duplex TTY streams can be used to read and write single characters at a time or groups of characters, optionally with echo turned off.

2.3.3. Reading Interrupt Characters from the TTY

On some platforms, the TTY stream may support the ability to read all interrupt and flow control characters as normal characters so that a terminal emulator can be written. This mode is controlled by bits passed to $readStream. Refer to Chapter 10 for a description of these bits.

2.3.4. Opening Serial TTY Lines

The basic TTY stream, $tty, is pre-opened by the STREAMS package and is connected to the "primary" input/output device, usually an interactive terminal. The ability to open additional serial TTY lines by name is supported on many systems. The program must provide an installation-dependent name that corresponds to the desired serial line (see Chapter 10).

2.3.5. Setting the Baud Rate on TTY Streams

The ability to set the baud rate on TTY streams is supported on some systems. If this capability is not supported, the procedures used to set the baud rate in Section 10.4 give error returns.

2.3.6. Writing a BREAK on TTY Streams

The ability to write a BREAK (something like a sustained NUL; often interpreted by programs as an interrupt or closed connection) on TTY streams is supported on some systems using the procedure described in Section 10.4. The procedure gives an error return if the capability is not supported.

2.3.7. Clearing Pending I/O

The ability to clear pending input and/or output is supported on some systems using the procedure described in Section 10.4. The procedure gives an error return if the capability is not supported.

2.3.8. Scheduling of TTY Streams

TTY streams may support scheduled I/O so that more than one coroutine doing TTY I/O may run at once (see below).

A program may determine if scheduling applies to the TTY stream on the host system, that is, if $tty can participate in scheduled I/O activities along with other streams, by the test "$isScheduled($tty)". If not, I/O to $tty must be assumed to block.

2.3.9. Catching Keyboard Interrupts

The TTY stream may support the ability to intercept and test a single keyboard interrupt character (CTRL-C on many systems). A program may determine whether catching TTY keyboard interrupts is possible on its $tty by calling the $enableInterrupts procedure and testing its return value.

If $enableInterrupts succeeds, the exception $keyboardInterruptExcpt is raised whenever the Scheduler gets control and discovers an interrupt that has not previously been observed. This exception is raised in the coroutine that most recently called $enableInterrupts.

If the $tty is scheduled, i.e., "$isScheduled($tty)" is true, the exception $keyboardInterruptExcpt can be raised while the program is waiting for TTY input. Otherwise, interrupts that occur during TTY input cause the exception to be raised the next time that the Scheduler is invoked following the TTY input.

2.3.10. Server/Client Communication

A platform may support server/client communication through one or more network protocols.

2.3.11. Process Rendezvous

A platform may support the ability of two processes to establish communication with each other, or rendezvous.

2.3.12. Child Process Creation

If child process creation is supported, the stream module SOCPRO is available for the system.

Child processes may be categorized as either cooperating or non-cooperating.

A child process that knows it is being invoked by a parent and that uses an agreed-upon communication protocol is called a cooperating child process. Typically, a cooperating child process uses its $tty for error messages and debugging and a secondary stream (available as the STREAMS system variable $parent) for communication with the parent process.

A child process that does not know that it is being run from a parent is called a non-cooperating child process. All communication with non-cooperating children takes place through the child's TTY.

Some systems support the ability to create non-cooperating child processes with a $tty that has all of the semantics normally associated with a $tty, including echo and line editing on input. On such systems the stream module called PTYPRO is available.

Limits on the ability to run non-cooperating general child processes through PTYPRO are system-dependent. On some systems, the PTYPRO stream is half-duplex, capable only of line-at-a-time I/O. The results of running a display editor in such a child process would be less than satisfactory!

2.3.13. The System Shell as a Child Process

The ability to run the standard system shell through PTYPRO is system-dependent. Even if the system supports the concept of a "shell", the results of running it may be less than satisfactory for some applications. The VAX/VMS shell, for example, turns off all prompting when run as a child process.

2.3.14. Scheduling of Advanced STREAMS

Scheduling of advanced STREAMS is available on some platforms. Without scheduling, the functionality of advanced STREAMS is limited because multiple I/O tasks cannot be performed simultaneously. Without scheduling, for example, servers can process only one client at a time and they can accept clients using only one network protocol.

The STREAMS system macro $systemSupportsScheduling tells whether the host system supports scheduling of advanced STREAMS. This support is independent of whether the system supports scheduling on TTY streams. Thus, there are four possible levels of scheduling possible on a given system:

```
IF $systemSupportsScheduling AND $isScheduled($tty) THEN
    # the system supports scheduling on all streams
EF $systemSupportsScheduling AND NOT $isScheduled($tty) THEN
    # the system supports scheduling of advanced STREAMS but
    # not $tty
EF $isScheduled($tty) AND NOT $systemSupportsScheduling THEN
    # the system supports scheduling of only $tty
EL # the system does not support scheduling at all
```

2.3.15. Timeouts

If a system supports scheduling then it may also support timeouts for stream-related activities. If a system supports timeouts in general then it supports timeouts for all scheduled streams (including TTY streams if they are scheduled).

The STREAMS system macro $systemSupportsTimeout tells whether the host system supports the ability to use timeouts on scheduled streams. If the TTY can be scheduled and $systemSupportsTimeout is true, timed input may also be performed on the TTY. If $systemSupportsTimeout is false, timeout values are ignored.

Currently, all systems that support scheduled streams also support timeouts.

2.4. Stream Types

The basic stream I/O procedures work uniformly regardless of the underlying stream device being used. Some stream types, most notably TTY streams, require special additional procedures and have limitiations specific to the device. These limitations must usually be understood by the programmer in order to use these devices effectively, except in simple cases.

The types of streams currently known to the system are:

- TTY streams
- socket (interprocess communication or IPC) streams
- memory streams
- PTY streams

TTY streams allow the program to communicate with a terminal. The terminal may be either a screen and a keyboard or a serial line (e.g., RS-232). For historical reasons that are not likely to change soon, TTY streams have special modes, limitations, and job control side effects.

Socket streams provide conceptually simple, unencumbered communication between processes.

Memory streams are a special kind of stream for internal communication between coroutines of the same process. They are created in pairs.

PTY streams are used to communicate with a child process that thinks it is communicating with a real terminal. The parent, which is typically a shell or a windowing system, passes on the

child's output, read from the PTY, to a terminal or window. The parent passes keystrokes to the child by writing to the PTY.

Other stream types may be implemented in the future.

2.5. Automatic Scheduling of Stream I/O

Task scheduling is crucial to distributed computing, since it allows a single process to interact with several other processes (and possibly also a user) without constraining the order in which the communications take place. The power of stream I/O for multiprocessing and multitasking lies in a special STREAMS system coroutine called the Scheduler.

Whenever any coroutine calls a stream I/O procedure, the Scheduler coroutine is resumed to decide what to do next. The original caller coroutine is resumed by the Scheduler only when its requested I/O is ready, i.e., when the I/O can be performed without blocking. In this way, the program runs much the way it would if each of its coroutines were run as a separate process under a multiprocess operating system, or as separate programs on two different computers.

Chapter 7 describes the use of stream scheduling for implementing multitasking programs.

2.6. Loose Coupling of Streams to MAINSAIL

In the current release of MAINSAIL, the stream I/O facilities are "loosely coupled" with the traditional MAINSAIL runtime system. In particular, unless special steps are taken, normal TTY I/O through cmdFile and logFile, the file "TTY", ttyRead and ttyWrite, and $timeout are performed independently of the stream Scheduler and therefore block.

Eventually it may be that all I/O to the file "TTY" (e.g., the default cmdFile and logFile) will go through the $tty stream, at least on systems where STREAMS is installed. The procedures ttyRead, ttyWrite, and ttycWrite may also eventually go through the $tty stream. Should this "strong coupling" be implemented, the procedure $timeout will work through the Scheduler if $systemSupportsTimeout is true.

In the meantime, to aid programs that wish to make output to the file "TTY" be scheduled, the device module NTTY is supplied. Output to a file opened through NTTY goes through $tty. In particular, if cmdFile and logFile are reopened through NTTY, any I/O to the terminal through them is scheduled, including error messages reported by errMsg. Example 2.6-1 shows how a program can arrange to use the NTTY device module in place of the file "TTY".

```
RESTOREFROM "strhdr";

POINTER(textFile) origCmdFile,newCmdFile,origLogFile,
    newLogFile;

INLINE PROCEDURE startup;
BEGIN
# Make cmdFile/logFile go through ntty>
enterLogicalName("tty","ntty" & $devModBrkStr);
setModName("tty","ntty");
origCmdFile := cmdFile;
origLogFile := logFile;
open(newCmdFile,"ntty" & $devModBrkStr,input);
open(newLogFile,"ntty" & $devModBrkStr,output);
cmdFile := newCmdFile; close(origCmdFile);
logFile := newLogFile; close(origLogFile);
END;



INLINE PROCEDURE cleanup;
BEGIN
# Restore original cmdFile/logFile
enterLogicalName("tty","");
setModName("tty","");
close(cmdFile); # Reopens to TTY
newLogFile := logFile;
open(origLogFile,"tty",output);
logFile := origLogFile;
close(newLogFile);
END;
```

Example 2.6-1. Using the NTTY Device Module (continued)

```
INITIAL PROCEDURE;
BEGIN
$initializeStreams;
startup;
...
END;


FINAL PROCEDURE;
cleanup;
```

Example 2.6-1. Using the NTTY Device Module (end)

3. Opening and Closing Streams

A stream is a two-way communication mechanism through which text and data may be transmitted. The other end of the stream may be an external device such as a keyboard and screen, another process, or a coroutine running as part of the same process.

Because streams are a fairly low-level concept, beginning programmers typically do not use streams directly. They may, however, use features implemented using stream I/O, such as the Remote Procedure Call facility or servers built by XIDAK.

3.1. Making STREAMS Available to a MAINSAIL Program: the STRHDR Intmod and $initializeStreams

In the current version of MAINSAIL, the procedures and symbols associated with STREAMS are made available to a MAINSAIL module by restoring from the STRHDR intmod when a module that uses STREAMS is compiled. It is also necessary at runtime to call the STREAMS system procedure $initializeStreams (shown in Table 3.1-1) in the module's initial procedure before making use of any other STREAMS facility (the effect is undefined if a STREAMS facility is used before $initializeStreams is called). $initializeStreams may be called more than once with no ill effects.

```
PROCEDURE   $initializeStreams;
```

Table 3.1-1. $initializeStreams

STREAMS identifiers may at some point be included in the MAINSAIL system intmod, in which case the restore from STRHDR and perhaps the call to $initializeStreams would become unnecessary.

3.2. Automatically Opened Streams: $tty and $parent

The stream $tty is automatically opened to the controlling terminal. Programs that wish to take advantage of the stream nature of $tty, to read single characters and control echo on full-duplex systems, for example, may do their I/O directly on $tty.

```
BEGIN "mod"

RESTOREFROM "strhdr";

...

INITIAL PROCEDURE;
BEGIN
$initializeStreams;
...
END;

END "mod"
```

Example 3.1-2. Use of $initializeStreams

When a cooperating child process is started by a MAINSAIL parent, the stream $parent is automatically opened to the parent process. The $parent stream is typically used for communication with the parent, while the $tty is used for I/O outside of the normal communications protocol, e.g., for error messages. If the process was not started as a cooperating process by its parent, the variable $parent is nullPointer.

3.3. The Class $stream

Stream records have the user-visible interface fields shown in Table 3.3-1.

The $name field is the full name used to open this stream (analogous to the name field of a file record). The $lastInputError and $lastOutputError fields contain the most recent error message associated with stream input and output. If the operation producing the error was neither input nor output, the message is placed in $lastInputError.

The system attribute macros describe the functionality of the underlying implementation of the stream. $systemSupportsScheduling and $systemSupportsTimeout apply to all streams except, possibly, the $tty stream. If the system does not support timeouts, timeout values other than $block are ignored (see Section 9.1.1).

```
CLASS $stream (
    STRING
        $name,
        $lastInputError,
        $lastOutputError;
        # Other user-visible fields are set according
        # to the stream type
);

BOOLEAN
<macro>     $systemSupportsScheduling;

BOOLEAN
<macro>     $systemSupportsTimeout;

BOOLEAN
<macro>     $isScheduled
                        (POINTER($stream) st);

BOOLEAN
<macro>     $isatty     (POINTER($stream) st);

BOOLEAN
<macro>     $isHalfDuplex
                        (POINTER($stream) st);
```

Table 3.3-1. User-Visible Fields of $stream and STREAMS Macros

The stream-specific attribute macros describe the functionality of a particular stream. The macro $isScheduled tells whether I/O to the stream st will be scheduled (see Chapter 7). If this macro returns false, the program may block until the I/O is complete.

Programmers should assume that all the attribute macros shown in Table 3.3-1 are evaluated at runtime rather than compiletime. On some platforms they may actually be evaluated at compiletime, but this is subject to change.

The stream-specific attribute macro $isHalfDuplex applies to TTY and PTYPRO streams only. It indicates that the terminal or pseduo-terminal is half-duplex. Note that testing the MAINSAIL system variable $attributes for the $halfDuplex bit applies only to the standard system primary input and output associated with a program. Other TTY streams (e.g., one

opened to a remote system) might not have the same characteristics as the primary input and output.

3.4. Opening and Closing Streams: $openStream and $closeStream

```
NOTE:  The actual strings passed to $openStream are
subject to change in subsequent releases of STREAMS.
Programmers should write their programs in a way that
will make it easy to accommodate this change.
```

```
BOOLEAN
PROCEDURE    $openStream (PRODUCES POINTER($stream) st;
                          STRING streamName;
                          OPTIONAL BITS ctrlBits;
                          PRODUCES OPTIONAL STRING
                              errorMsg);

BOOLEAN
PROCEDURE    $openStream (PRODUCES POINTER($stream)
                              st1,st2;
                          STRING streamName;
                          OPTIONAL BITS ctrlBits;
                          PRODUCES OPTIONAL STRING
                              errorMsg);

BOOLEAN
PROCEDURE    $closeStream
                         (MODIFIES POINTER($stream) st;
                          OPTIONAL BITS ctrlBits);
```

Table 3.4-1. $openStream (Generic) and $closeStream

Streams are opened and closed by explicit calls that manipulate a $stream record.

$openStream opens a stream identified by streamName, making it available for I/O in accordance with ctrlBits.

The stream name is of the form:

```
strmod>name
```

or:

```
strmod(host)>name
```

where strmod is the name of a stream module (or special stream prefix; this prefix is often called a "stream module" in the documentation even when it is a special name recognized by $openStream instead of the name of an actual MAINSAIL module), name is a name with a syntax and meaning that depends on the stream module name, and host (if supplied) is the host system on which the stream is to be opened.

The character ">" is always used as the stream prefix separator, unlike the corresponding separator for file names. The file name separator, $devModBrk, is ">" on most operating systems but may differ on some systems. The stream name separator is always ">", so no special identifier is provided for it; programs that specify stream names should just use the character ">" in the strings specifying the names.

The description given in this document of each stream module gives the naming conventions and characteristics of each stream module.

Some stream modules produce only one stream when an $openStream is done. Such stream modules include those that communicate with an external device, such as TTYSTR. For these stream modules, the first form of $openStream must be used.

Some stream modules, such as MEMSTR, always produce two streams. Stream pairs allow different coroutines within the same process to communicate with each other. For these stream modules, the second form of $openStream must be used.

Some stream modules can produce either one or two streams. For example, SOCPRO and PTYPRO produce one stream to talk to the child process's $tty, and may optionally produce another stream to talk to the child's $parent. For these stream modules, either the first or second form of $openStream may be used.

If $openStream is able to open the stream, it produces the stream handle record(s) and returns true.

The valid ctrlBits bits are input, output, and errorOK. If neither of input or output is given, both are assumed. In the second form of $openStream to a memory stream, specifying either input or output, but not both, makes st1 work in the specified direction and st2 work in the opposite direction.

```
errorOK not set:   errMsg is called and a new stream name
                   is reprompted for until a valid name is
                   given.  A valid stream handle is always
                   returned, and $openStream always returns
                   true.

errorOK set:       errorMsg is set to a description of the
                   error, the stream handles to
                   nullPointer, and $openStream returns
                   false.
```

Table 3.4-2. Error Handling in $openStream

The procedure $closeStream closes a stream and sets its pointer argument to nullPointer. In the case of stream pairs opened with $openStream, both pointers should be closed with $closeStream. After a stream is closed, no further I/O may be performed on the stream.

If an error occurs during closing (e.g., the stream handle is invalid) and errorOK is given in ctrlBits, $closeStream returns false after printing the error message; if errorOK is not set, a fatal error occurs.

4. Remote Modules and Remote Procedure Calls (RPC)

The remote module binding mechanism is a high-level interface that simplifies the construction of algorithms that consist of multiple cooperative processes. The remote module facility handles the required communication automatically through what look like ordinary procedure calls, allowing the programmer to concentrate on solving the application problem.

Remote modules are a high-level concept built using the interprocess communication methods described in Chapter 6. The idea is to make an interaction with another process look like the creation of a new MAINSAIL module instance (the "remote module") in the remote process. Intermodule procedure calls to the remote module are called "remote procedure calls" (RPC).

The remote module may be created in any process with which a program is communicating using STREAMS. Commonly, the remote module is created in either a global server process, in a child process, or in a parent process.

In the discussion that follows, the process in which the remote module actually runs is often termed the "RPC server", and the process that makes the calls the "RPC client". However, these processes need not be servers and clients in the senses given in Section 6.3. The server may be thought of as the "callee process" and the client as the "caller process".

RPC clients may be written in MAINSAIL or in C. The MAINSAIL case is described first, as many of the MAINSAIL client concepts are applicable to C as well. RPC servers must be written in MAINSAIL.

To create a new instance of a remote module, a MAINSAIL program must know how to open a stream that communicates with the process that is providing that remote module, or else it must already have a stream that is open to that process. In the case of a remote module provided by a global server, the user needs to know the name of the service and possibly the host system, if the service is provided on multiple systems.

4.1. Overview

To provide remote computation using RPC, the programmer writes a MAINSAIL module (the server) with interface procedures that provide the desired computation. The module's interface declaration must use the prefix class $remoteModuleCls. The remote module is compiled with the "RPC" subcommand (with no arguments) to the MAINSAIL compiler to produce two MAINSAIL source modules. The source module names are derived from the remote module

name by appending "SRV" and "CLI" to the first three letters of the remote module name, and the output file names are derived from the module names by converting them to lower case and adding ".msl". For example, if the remote module is FOOMOD, the compilation produces the two MAINSAIL source modules FOOCLI and FOOSRV in the files "foocli.msl" and "foosrv.msl".

In the case of a remote module FOOMOD, the FOOMOD and FOOSRV modules are compiled for the system that is to execute the remote module, and FOOCLI is compiled for the system that will call the remote module.

Figure 4.1-1 shows a process A creating a new instance of the remote module FOOMOD and calling the procedure proc1 in FOOMOD.

The module FOOCLI is a "stub" that runs in the "client" process. It has the same interface procedures as FOOMOD and it implements those procedures by communicating with the "server" process, sending and receiving the arguments as necessary, and returning the values produced by the procedure call.

The module FOOSRV is the companion stub to FOOCLI that runs in the server process. It communicates with FOOCLI to receive the arguments, then it invokes the correct procedure in the "real" module, FOOMOD, to do the actual work. It sends back the resultant modified arguments to FOOCLI.

The remote module itself, FOOMOD, is written much as any other MAINSAIL module would be written, except that certain restrictions on the argument types of procedures apply, argument transmission efficiency considerations must be taken into account, its interface declaration must use prefix class $remoteModuleCls, and no module interface variables are allowed.

When developing distributed applications, it is often useful to write the module that will eventually be remote (e.g., FOOMOD) as a local module and debug it that way. Then, to distribute the application, the remote module can be compiled with the RPC compiler and either installed into a server or made available to the process that is to provide it.

The call to $newRemoteModule in the process A actually creates a new instance of the module FOOCLI in A's address space. Recall that FOOCLI was produced by the RPC compiler from the interface of the "real" FOOMOD module. In its initial procedure, FOOCLI uses STREAMS to connect to the service "abc" in the process B. B creates a new instance of the FOOSRV module. FOOSRV in turn creates a new instance of the FOOMOD module. The connection is maintained until the instance of FOOMOD is disposed.

When the process A makes the intermodule call FOOCLI.proc1, FOOCLI.proc1 communicates with FOOSRV in B, telling it that "proc1" is being called. FOOSRV in B invokes FOOSRV.proc1 to read the uses arguments from FOOCLI.proc1. FOOSRV.proc1 then calls

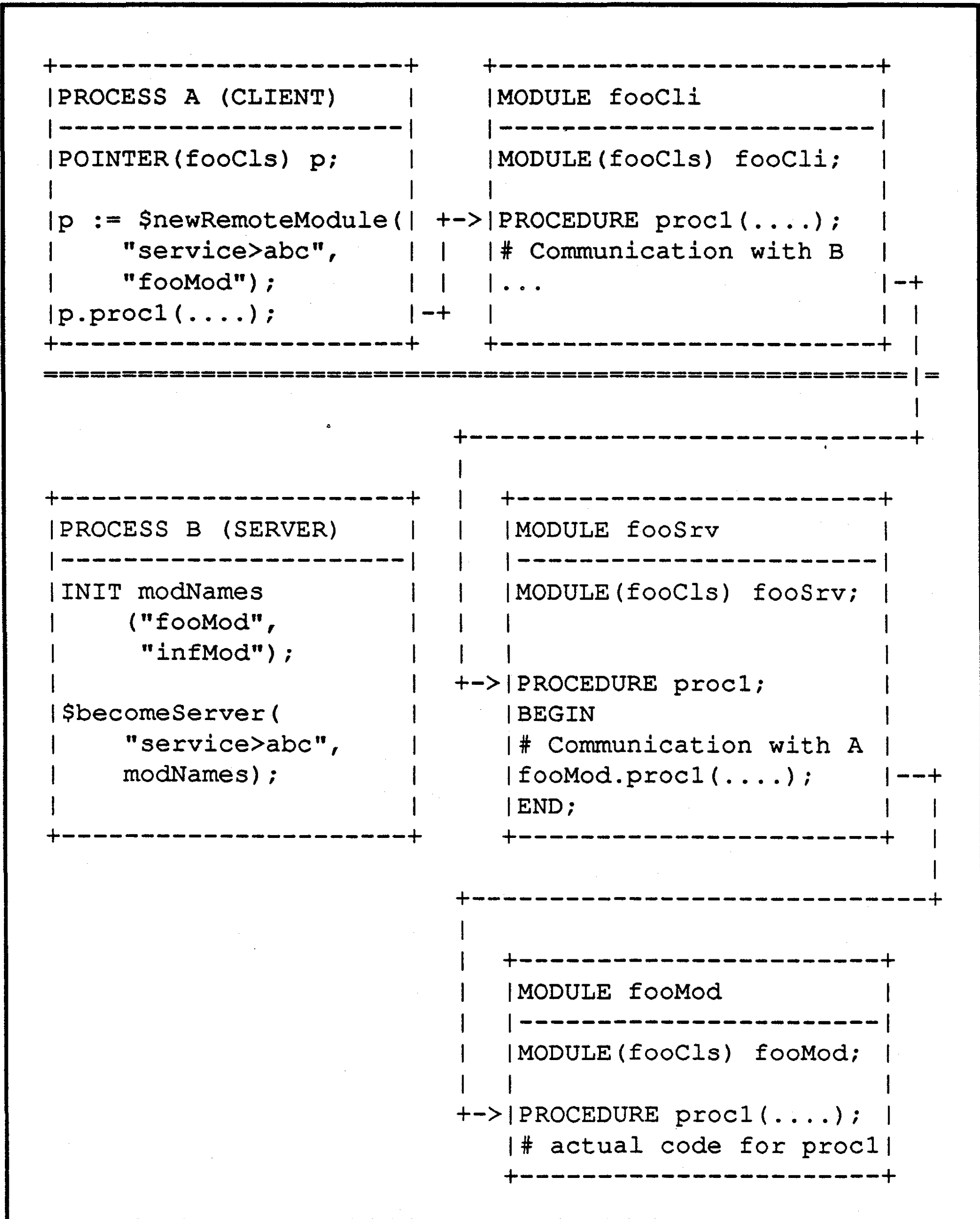


Figure 4.1-1. Schematic Diagram of a Remote Procedure Call

the "real" proc1, FOOMOD.proc1, and communicates the results back to FOOCLI.proc1, which returns them to the original caller in A.

During the call to proc1, the current coroutine in A is blocked, just as it would be for a local procedure call. Because STREAMS communication is used, other scheduled coroutines in A, if any, can run while FOOCLI.proc1 is waiting for the results from FOOSRV.proc1.

In the example above, B is a global server process that uses the $becomeServer procedure to become an RPC server that provides the modules FOOMOD and INFMOD under the service name "abc". The $becomeServer call does not return until the server is killed.

4.2. Remote Module Procedures

```
POINTER($remoteModuleCls)
PROCEDURE   $newRemoteModule
                        (STRING serviceStreamName;
                         STRING moduleName;
                         OPTIONAL LONG INTEGER cliVersion,
                             oldestSrvVersion;
                         OPTIONAL BITS ctrlBits;
                         OPTIONAL PRODUCES STRING msg);

POINTER($remoteModuleCls)
PROCEDURE   $newRemoteModule
                        (POINTER($stream) st;
                         STRING moduleName;
                         OPTIONAL LONG INTEGER cliVersion,
                             oldestSrvVersion;
                         OPTIONAL BITS ctrlBits;
                         OPTIONAL PRODUCES STRING msg);

BOOLEAN
PROCEDURE   $becomeServer
                        (STRING servicePortName;
                         STRING ARRAY(*) moduleNames;
                         OPTIONAL BITS ctrlBits;
                         OPTIONAL PRODUCES STRING msg);
```

Table 4.2-1. Remote Module Procedures (continued)

```
BOOLEAN
PROCEDURE    $becomeServer
                        (POINTER($stream) st;
                         STRING ARRAY(*) moduleNames;
                         OPTIONAL BITS ctrlBits;
                         OPTIONAL PRODUCES STRING msg);
```

Table 4.2-1. Remote Module Procedures (end)

$newRemoteModule creates a new remote instance of the module named moduleName. It returns a pointer to a new (unbound) instance of the client end of the desired module. If errorOK is set in ctrlBits, errors cause a nullPointer return with msg set; otherwise, a fatal error message is issued. The use of optional version numbers is discussed below.

The first instance of $newRemoteModule takes a stream name that can be used to establish a connection with the desired server process or create the desired child process. It calls $openStream to open serviceStreamName and creates a new local instance of moduleName to talk to a new remote instance of moduleName created in the server.

The second instance of $newRemoteModule takes an already open stream that is connected to a process that is about to call or has already called $becomeServer on the other end of the stream. It creates a local instance of moduleName to talk to a new remote instance of moduleName created by the process at the other end of the stream st. This form of $newRemoteModule is useful for talking to child or parent processes when the streams are already open.

$becomeServer becomes a server for the remote modules named by the elements of moduleNames. If errorOK is set in ctrlBits, errors in becoming a server cause a false return with msg set; otherwise, a fatal error message is issued.

The first form of $becomeServer sets up one or more ports on which new clients are accepted as they connect to the service of which the name is given as part of servicePortName. This form is used by a global server process. For each client that connects, a service coroutine is started to provide one of the remote modules to that client. The service coroutines run scheduled so that more than one client can be serviced at once. When the client disconnects, the service coroutine is killed automatically.

The port name may specify an explicit network protocol, e.g., "tcp>abc", or it may specify the generic protocol "service", e.g., "service>abc", to serve clients using every network protocol that the server is declared to serve. The service "abc" must have been previously declared as a valid service name in the service protocol table as described in Appendix A.

The second form of $becomeServer may be used by a process that wishes to provide a remote module to the process at the other end of the stream st. It provides any of the remote modules in moduleNames to the process at the other end of the stream st. The process on the other end of the stream must call $newRemoteModule to create a new remote module. When the process at the other end disposes of the remote module, the second form of $becomeServer returns with the stream still open.

In either case of $becomeServer, the server program can perform independent computation and/or stream I/O on its own in parallel by calling $becomeServer in an independent scheduled coroutine that it sets up.

4.3. Remote Module Interfaces

The following restrictions apply to module interface declarations for remote modules:

- The module interface must use the prefix class $remoteModuleCls.
- The module may have only interface procedures; no interface variables are allowed.
- The data types address and charadr are reserved for buffer passing (see below). The type of a procedure may not be address or charadr.
- Pointers and arrays must be the roots of structures that can be legally read and written in PDF structure images by the Structure Blaster. Data sections should not be included in structures that might be sent to a server running on another system or another version of MAINSAIL (if a data section is to be valid in the RPC server, the RPC server must have access to the appropriate objmod).
- The module must be written to operate as an unbound module.
- The macro $remoteModuleDefaults must be invoked at the end of the remote module (see Example 4.5-1). This macro defines procedure bodies for two interface procedures in the $remoteModuleCls prefix class.

The semantics of uses, modifies, and produces for pointers and arrays are slighty different between local calls and remote calls. In local calls (standard MAINSAIL), a produces or modifies pointer or array means that the pointer value itself can be changed. The structure referred to by the pointer or array can be changed even through a uses parameter. Since RPC involves a transfer of data, uses, modifies, and produces refer to the entire structure being pointed to; i.e., a uses pointer or array sends the structure to the remote module, but does not read it back. A produces pointer or array does not send a structure, but reads a structure back from the remote module. A modifies pointer or array both writes and reads the structure.

4.4. Buffer Passing Conventions

A buffer is a block of storage units, portable storage units (PDF data), or character units. The start of a buffer of storage units is given by an address parameter. The start of a buffer of portable storage units or characters is given by a charadr parameter. The data types address and charadr are thus reserved for buffer passing between the two processes.

Buffers containing characters are distinguished from buffers containing portable data by the name of the charadr parameter. Names ending with "pdf" (case is not distinguished in parameter names) indicate portable data; other names indicate characters. Characters are translated by the remote procedure call interface automatically, while PDF and storage unit data are not.

The length of the buffer is given in parameters following the address or charadr parameter. If the buffer contains storage units, the length and size are given in storage units; if the buffer contains characters or PDF data, the length and size are given in characters.

Like other procedure parameters, buffers may be uses, modifies, or produces. A uses buffer passes the data in the buffer to the procedure. A produces buffer receives data from the procedure. A modifies buffer passes data to the procedure and receives data back (the returned data overwrite the original data).

For all three kinds of buffers, the address or charadr parameter must itself be declared as a uses parameter. The caller of the procedure must thus supply a valid address or charadr; i.e., the parameter may not be nullAddress or nullCharadr.

A uses buffer is indicated by following the address or charadr by a single uses long integer parameter giving the amount of data supplied in the buffer. The parameter name must end with "length".

Produces and modifies buffers are indicated by following the address or charadr by two long integer parameters. The first must be a uses parameter giving the allocated size of the buffer. Its name must end with "size". For produces buffers, the second parameter must be a produces parameter that will be set to the length of data supplied by the procedure call. For modifies buffers, the second parameter must be a modifies parameter that is initially set to the length of data in the buffer and is modified to the length of data supplied by the procedure call. In both cases, the name of the second parameter must end with "length".

Example 4.4-1 shows an example module declaration of a remote module with uses, produces, and modifies buffer parameters for storage unit, character, and portable data buffers.

```
MODULE($remoteModuleCls) xxx (

    # USES Buffers

    PROCEDURE   writeDataBuf
                            (ADDRESS buf;
                             LONG INTEGER bufLength);
    PROCEDURE   writeTextBuf
                            (CHARADR buf;
                             LONG INTEGER bufLength);
    PROCEDURE   writePdfBuf (CHARADR bufPdf;
                             LONG INTEGER bufLength);

    # PRODUCES Buffers

    PROCEDURE   readDataBuf (ADDRESS buf;
                             LONG INTEGER bufSize;
                             PRODUCES LONG INTEGER
                                 bufLength);
    PROCEDURE   readTextBuf (CHARADR buf;
                             LONG INTEGER bufSize;
                             PRODUCES LONG INTEGER
                                 bufLength);
    PROCEDURE   readPdfBuf  (CHARADR bufPdf;
                             LONG INTEGER bufSize;
                             PRODUCES LONG INTEGER
                                 bufLength);

    # MODIFIES Buffers
```

Example 4.4-1. Example Uses, Produces, and Modifies Buffer Parameters (continued)

```
    PROCEDURE    writeAndReadDataBuf
                              (ADDRESS buf;
                               LONG INTEGER bufSize;
                               MODIFIES LONG INTEGER
                                   bufLength);
    PROCEDURE    writeAndReadTextBuf
                              (CHARADR buf;
                               LONG INTEGER bufSize;
                               MODIFIES LONG INTEGER
                                   bufLength);
    PROCEDURE     writeAndReadPdfBuf
                              (CHARADR bufPdf;
                               LONG INTEGER bufSize;
                               MODIFIES LONG INTEGER
                                   bufLength);

);
```

Example 4.4-1. Example Uses, Produces, and Modifies Buffer Parameters (end)

4.5. Example of Remote Modules between a Parent and Child Process

Example 4.5-1 shows an example remote module that provides a single trivial remote procedure "hello".

Examples 4.5-2 and 4.5-3 are an example of the use of fooMod by executing it in a child and calling it from a parent.

In its initial procedure, the parent, shown in Example 4.5-2, creates a child process running the module CHILD under the current version of MAINSAIL. It sets up a coroutine to multiplex the child's TTY (for error messages and debugging) to the parent's TTY. It then creates a new instance of the remote module FOOMOD in the child and calls the procedure "hello" in the remote module. Eventually it disposes the remote module, and kills the child process.

Example 4.5-3 shows how the module CHILD might be written. It becomes a server for the remote module FOOMOD.

```
BEGIN "fooMod"

RESTOREFROM "rpchdr";

MODULE($remoteModuleCls) fooMod (
    STRING PROCEDURE hello (STRING s);
);

STRING PROCEDURE hello (STRING s);
RETURN("How do you do " & s);

$remoteModuleDefaults

END "fooMod"
```

Example 4.5-1. Example Remote Module

```
BEGIN "child"

RESTOREFROM "rpchdr";

CLASS($remoteModuleCls) fooModCls (
    STRING PROCEDURE hello (STRING s);
);

INITIAL PROCEDURE;
BEGIN
STRING ARRAY(1 TO 1) modNames;

new(modNames);
INIT modNames ("fooMod");

$becomeServer($parent,modNames);
END;

END "child"
```

Example 4.5-3. Child Providing the FOOMOD Remote Module

```
BEGIN "parent"

RESTOREFROM "rpchdr";

CLASS($remoteModuleCls) fooModCls (
    STRING PROCEDURE hello (STRING s);
);

POINTER($stream) chTty;
POINTER($coroutine) rc, wc;

PROCEDURE reader;
BEGIN
STRING s;
WHILE $success($readStream(chTty,s,errorOK)) DO
    $writeStream($tty,s,$line);
IF wc AND NOT $killedCoroutine(wc) THEN
    $killCoroutine(wc);
$reschedule(delete);
END;


PROCEDURE writer;
BEGIN
STRING s;
DO  $readStream($tty,s)
    UNTIL NOT $success($writeStream(chTty,s,errorOK));
IF rc AND NOT $killedCoroutine(rc) THEN
    $killCoroutine(rc);
$reschedule(delete);
END;


INITIAL PROCEDURE;
BEGIN
STRING s;
POINTER(fooModCls) m;
POINTER($stream) chCtl;
```

Example 4.5-2. Parent Executing Remote Module FOOMOD in a Child (continued)

```
# Create the child
$openStream(chTty,chCtl,"socpro>" & $executableBootName &
    " child");

# Handle its tty
rc := $createCoroutine(thisDataSection,"reader");
$queueCoroutine(rc);
wc := $createCoroutine(thisDataSection,"writer");
$queueCoroutine(wc);

# Use it to supply FOOMOD
m := $newRemoteModule(chCtl,"fooMod");
s := m.hello("xxx");
...
dispose(m);

# Kill the child (kills the reader and writer implicitly)
$closeStream(chTty); $closeStream(chCtl);
END;

END "parent"
```

Example 4.5-2. Parent Executing Remote Module FOOMOD in a Child (end)

A generic module RPCSRV (see Section 4.7) is supplied with the RPC package. The RPCSRV module provides the functionality of the child shown in Example 4.5-3 in a generic way.

4.6. Example Clients and Servers Using Remote Modules

Example 4.6-1 shows the same remote module as in Example 4.5-1 except that it declares its optional protocol version numbers.

Examples 4.6-2 and 4.6-3 are examples of a server program that provides FOOMOD, and a client program that executes FOOMOD remotely.

The client is written like the parent in the previous section, except that it asks to talk to the server named "foo" instead of creating a child process. The server process is written like the

```
BEGIN "fooMod"

RESTOREFROM "rpchdr";

INITIAL PROCEDURE;
BEGIN
newestVersion := 5L;        # OPTIONAL (see below)
oldestVersion := 3L;        # OPTIONAL (see below)
END;

MODULE($remoteModuleCls) fooMod (
    STRING PROCEDURE hello (STRING s);
);

STRING PROCEDURE hello (STRING s);
RETURN("How do you do");

$remoteModuleDefaults

END "fooMod"
```

Example 4.6-1. Example Remote Module

child in the previous section, except that it becomes a server to "service>foo" instead of to its parent.

The server shown in Example 4.6-3 keeps accepting and servicing new clients indefinitely.

4.6.1. Use of Version Numbers

When a remote module is provided as part of a service, the author of the service must provide writers of client modules with documentation that includes:

- the declaration for the remote module,
- a description of the semantics of the procedures in the remote module declaration, and

```
BEGIN "client"

RESTOREFROM "rpchdr";

CLASS($remoteModuleCls) fooModCls (
    STRING PROCEDURE hello (STRING s);
);

INITIAL PROCEDURE;
BEGIN
STRING s;
POINTER(fooModCls) m;

# Ask for version 4L of fooMod
m := $newRemoteModule("service>foo","fooMod",4L);
s := m.hello("xxx");
...
dispose(m);
END;

END "client"
```

Example 4.6-2. Example Client of the FOOMOD Remote Module Service

- a service version number identifying the version of the remote module to which the accompanying description applies.

The declaration of the remote module allows the client program to declare a pointer to the remote module's class so that procedure calls can be made using MAINSAIL's "p.f" construct.

The description of the semantics of the procedures allows the client to make use of the remote module to do useful work.

The service version number allows the client to identify the semantics it is expecting to the server. If a new version of the server is written that accepts new or additional semantics in addition to the old semantics, the server may be written so that it can also service older clients.

When a remote module is written for use in a server, it may set two version parameters in its initial procedure, as shown in Example 4.6.1-1.

```
BEGIN "server"

RESTOREFROM "rpchdr";

CLASS($remoteModuleCls) fooModCls (
    STRING PROCEDURE hello (STRING s);
);

INITIAL PROCEDURE;
BEGIN
STRING ARRAY(1 TO 1) modNames;

new(modNames);
INIT modNames ("fooMod");

$becomeServer("service>foo",modNames);
END;

END "server"
```

Example 4.6-3. Example Server Providing the FOOMOD Remote Module

```
BEGIN "fooMod"

...

INITIAL PROCEDURE;
BEGIN
oldestVersion := 3L;
newestVersion := 5L;
...
END;

END "fooMod"
```

Example 4.6.1-1. A Remote Module That Sets Its Version Numbers

These numbers describe the oldest and newest semantics the server is willing to provide to the client, thus allowing servers to be updated with new semantics while still providing the older semantics to older clients.

newestVersion is the current version of the remote module semantics. oldestVersion is the oldest backward-compatible version that the server provides.

If a server provides no version numbers, the default for both version numbers is 0L.

If the code in the remote module wishes to adapt to the client's actual version, it may access the long integer interface variable "version". This interface variable is declared in $remoteModuleCls, as shown in Table 4.6.1-2. It is set to the value of the cliVersion parameter in the call to $newRemoteModule.

```
CLASS $remoteModuleCls (
    ...
    LONG INTEGER version;
    ...
);
```

Table 4.6.1-2. User-Visible Fields of $remoteModuleCls

4.6.2. Writing Adaptable Clients

Version numbers allow new servers to be installed that support both new and old clients, and for new clients to make use of older servers. By giving an explicit remote module version number (or range of version numbers) in the $newRemoteModule call, the client can indicate which module version (or range of module versions) it is written to use.

When the server and client establish a connection at the time of the call to $newRemoteModule, the underlying communication code determines if there is an overlap between the versions provided by the server and the versions acceptable to the client, and if so, the highest common version number. This highest number is then made available to the "true" remote module in the long integer module interface variable "version".

A client may ask for a range of service version numbers when it calls $newRemoteModule. It may then obtain the actual remote module version provided by the server and adapt to the published semantic and syntactic specifications of the returned version.

If different versions provide different module interfaces, the client must not call any procedures of which the interface has changed (this requirement is enforced by runtime checking).

4.6.3. The Server Log File

When a server is started with the first form of $becomeServer, a log of the server's activities is kept in the file "server.log". By examining that file (in the directory on which the server is running), a user may determine what the server has been doing.

Entries are made automatically into "server.log" when the following activities occur in a port based server:

- The server is started with the first form of $becomeServer.
- A client connects and starts a remote module.
- A client disposes a remote module or disconnects.
- An unhandled error occurs in a remote module (the call stack is listed).

If multiple servers are run from the same directory, a MAINEX "ENTER" subcommand can be used in each server to cause the log files to have different names, if desired.

A sample "server.log" file is shown in Example 4.6.3-1.

```
29-Feb-88 14:21:
    Server service>srv Started, RPC 1 MAINSAIL 11.20
29-Feb-88 14:21:
    Client: bob@Poseidon.1 fooMod  5 3  29-Feb-88 14:21
29-Feb-88 14:32:
    Finish: bob@Poseidon.1 fooMod  5 3  29-Feb-88 14:21
```

Example 4.6.3-1. Example "server.log" Log File

4.6.4. Terminating a Global Server

When the first form of $becomeServer is used, the server process can be killed by raising the exception $killServerExcpt in one of the remote modules. This exception causes

$becomeServer to return. $becomeServer cleans up after itself by closing any streams that it opened.

4.7. Generic RPC Server: the RPCSRV Module

The RPCSRV module is provided by the RPC package to facilitate the creation of child and server processes for remote modules. RPCSRV is a generic module that provides remote modules either to its parent or as a global server.

RPCSRV accepts a command line of the form:

```
rpcsrv <servicePortName> rm1 rm2 rm3 ...
```

to start a global server on the port named <servicePortName>, or:

```
rpcsrv - rm1 rm2 rm3 ...
```

to start a parent server on $parent. In both cases, the names rm1, rm2, etc., are names of remote modules to supply to the clients or parent.

Using the RPCSRV module, a server named "foo" that provides the RPC modules "fooMod" and "infMod" can be started using the UNIX shell command:

```
/mslDirectory/mainsa rpcsrv service\>foo fooMod infMod
```

where "mslDirectory" is the MAINSAIL directory. Note that under the UNIX shell, you must quote the ">" used in the service port name with the shell escape character "\".

Using the RPCSRV module, a child process that provides the RPC module FOOMOD can be started using the $openStream procedure in a MAINSAIL program:

```
$openStream(chTty,chCtl,"socpro>" & $executableBootName &
            " rpcsrv - fooMod");
```

4.8. RPC Implementation Restrictions

Only one $newRemoteModule may be active on a given stream, in the case that the second form of $newRemoteModule is called. This restriction can be lifted at some future date, if it becomes a nuisance.

A server process must do a $becomeServer before a client process can call $newRemoteModule to talk to that server. This restriction is not severe because server processes are usually started when the node on which they run comes up.

4.9. RPC Efficiency Considerations

Because many network protocols are often slow, remote modules should be designed to handle relatively large subtasks rather than trivial ones. When possible, all the necessary uses and produces data should be passed in a single call rather than multiple calls.

For example, using TCP on a SUN 3/60, the wall clock time to bind a remote module varies from a fraction of a second to seconds, depending on whether the server process is paged out. Once the remote module is bound, the wall clock time to complete a trivial remote procedure call is typically about 0.2 seconds. This time is almost entirely due to TCP communications overhead.

We believe that the overhead per procedure call could be improved with a faster UNIX network protocol such as UDP. In addition, on operating systems that support faster mechanisms, the overhead may be considerably less. However, if the remote procedure does a non-trivial amount of computation (e.g., several seconds' worth), even the communication overhead of TCP is not a significant factor.

4.10. Parallel Processing Using RPC Calls

Section 7.7.2 describes how RPC calls can be performed in parallel using multitasking.

4.11. Remote Exceptions

```
TEMPORARY FEATURE:  SUBJECT TO CHANGE
```

An experimental temporary feature has been added to RPC in the ability to propagate exceptions from a remote module to the calling client, including the ability to examine the calls stack and to debug the server.

The "model" for exception handling is that the remote procedure execution is taking place in the client, so exceptions go up to the procedure invocation in the server stub, then are raised in

the client stub. Certain exceptions are considered local to the server (mainly $abortProcedureExcpt). These are not raised in the client.

$exceptionPointerArg is always set to nullPointer in the client, since it would be undesirable to pass a large structure when an exception occurred.

$systemExcpt is handled specially. If a remote procedure call causes a $systemExcpt, errMsg is called in the client with some extra registered exceptions: "remote calls" to see the remote calls stack, "remote PRNTCO" to see the output of PRNTCO run in the server, and "remote debug" to debug the server. $exceptionStringArg1 is prefixed with "RPC:" when $systemExcpt is raised, so that "RPC:" appears in any error message displayed to the user.

Debugging the server currently means that a call to $debugExec is made in the server. If the server is running on a terminal or terminal emulator, it can be debugged. This is very useful (compared to the way things have been up until now) but one could imagine other debugging mechanisms.

4.12. C RPC

```
The C RPC facility is still being designed,
and so is especially likely to change.
```

RPC clients written in C are similar to those written in MAINSAIL except:

- The server (written in MAINSAIL) must be compiled with the "RPC C" subcommand (i.e., the "RPC" subcommand followed by the argument "C"). This produces a header file and a client stub procedure file; the latter is analogous to the FOOCLI module in Figure 4.1-1. The base of the two file names is derived from the remote module name by appending "cli" to the first three letters of the remote module name converted to lower case. The header file has the extension ".h" and the client stub procedure file the extension ".c"; for example, "foocli.h" and "foocli.c" would be produced for a remote module FOOMOD.

- The RPC client source file must include (with "#include" directives) a file "mrpc.h", which is shipped with the MAINSAIL STREAMS package, and, for each remote module in which it calls a procedure, the header file generated when the remote module was compiled with the "RPC C" subcommand.

- Instead of opening a stream to the RPC server, the client calls the procedure "connserver" (provided by the file "mrpc1.o", which resides on the MAINSAIL directory).

- Instead of calling $newRemoteModule, the C client calls a procedure called by the name of the remote module (in lower case) with "_init" added to it, e.g., "foomod_init", with the int value returned by connserver as one of the arguments, to initialize the connection. This special procedure is called the remote module's _init procedure.

- The remote procedures, instead of being called with a direct pointer call, as in MAINSAIL, are called with the procedure name converted to lower case and prefixed by the remote module name and an underscore, e.g., "foomod_proc1", and with the value returned from the _init procedure as the first argument.

- When the connection is to be closed, the C program calls a _final procedure, analogous to the _init procedure, instead of $closeStream, and "close" must be called for the value returned from connserver.

- MAINSAIL server procedures with pointer arguments or return values cannot be called. Array and string arguments must be handled specially as described below.

- The client must be linked with the client stub procedures and with the file "mrpc1.o" from the MAINSAIL directory.

The correspondence between C client arguments and MAINSAIL remote module parameters is shown in Table 4.12-1.

4.12.1. Data Type Rules

String arguments must be null-terminated. Strings containing the character code 0 cannot be sent to or received from the remote procedure. Modifies and uses strings returned by remote procedures must be passed to cfree to reclaim the space they occupy. If the string or array passed to the remote procedure was allocated with malloc, then the client is also responsible for deallocating it by passing it to cfree; it is not automatically deallocated by RPC after transmission.

Arrays are passed as pointers to a C struct ARRAY typedefed in "mrpc.h" as:

MAINSAIL		C	NOTES
USES	BOOLEAN	short	0 is false,
MODIFIES	BOOLEAN	short *	else true
PRODUCES	BOOLEAN	short *	
USES	INTEGER	short	
MODIFIES	INTEGER	short *	
PRODUCES	INTEGER	short *	
USES	LONG INTEGER	long	
MODIFIES	LONG INTEGER	long *	
PRODUCES	LONG INTEGER	long *	
USES	REAL	float	
MODIFIES	REAL	float *	
PRODUCES	REAL	float *	
USES	LONG REAL	double	
MODIFIES	LONG REAL	double *	
PRODUCES	LONG REAL	double *	
USES	BITS	short	Same as INTEGER
MODIFIES	BITS	short *	
PRODUCES	BITS	short *	
USES	LONG BITS	long	Same as LONG
MODIFIES	LONG BITS	long *	INTEGER
PRODUCES	LONG BITS	long *	
USES	STRING	char *	
MODIFIES	STRING	char **	Caller must cfree result
PRODUCES	STRING	char **	Caller must cfree result
USES	ARRAY	ARRAY *	allocated by caller
MODIFIES	ARRAY	ARRAY *	return allocated by RPC
PRODUCES	ARRAY	ARRAY *	allocated by RPC

Table 4.12-1. C Client Arguments and MAINSAIL Server Parameters (continued)

```
POINTER                            forbidden

USES     buffer                    char *,
  ADDRESS, LONG INTEGER            long
  CHARADR, LONG INTEGER
MODIFIES buffer                    char *,
  ADDRESS, LONG INTEGER,           long,
    MODIFIES LONG INTEGER          long *
  CHARADR, LONG INTEGER,
    MODIFIES LONG INTEGER
PRODUCES buffer                    char *,
  ADDRESS, LONG INTEGER,           long,
    PRODUCES LONG INTEGER          long *
  CHARADR, LONG INTEGER,
    PRODUCES LONG INTEGER
```

Table 4.12-1. C Client Arguments and MAINSAIL Server Parameters (end)

```
typedef struct mainsail_array_descr {
    char    *ary_first_elem;
    long    ary_lb1,ary_ub1,
            ary_lb2,ary_ub2,
            ary_lb3,ary_ub3;
    short   ary_dims, ary_type;
    long    ary_alloc_status;
    } ARRAY;
```

When a C procedure passes an array to a remote procedure, it must first set all the relevant fields of the ARRAY struct. The meanings of the fields are shown in Table 4.12.1-1. Since MAINSAIL arrays, unlike C arrays, are not necessarily zero-origin, lower bounds as well as upper bounds of each dimension must be specified.

The ary_type field is the data type of the MAINSAIL array. It should be set to one of the following constants defined in "mrpc.h":

Field	Meaning
ary_first_elem	Pointer to first element
ary_dims	Number of dimensions (1, 2, or 3)
ary_type	MAINSAIL data type
ary_lb1	Lower bound of first dimension
ary_ub1	Upper bound of first dimension
ary_lb2	Lower bound of second dimension (only if ary_dims GEQ 2)
ary_ub2	Upper bound of second dimension (only if ary_dims GEQ 2)
ary_lb3	Lower bound of third dimension (only if ary_dims = 3)
ary_ub3	Upper bound of third dimension (only if ary_dims = 3)
ary_alloc_status	For use with dispose_array

Table 4.12.1-1. ARRAY Struct Fields and Meanings

MAINSAIL Data Type	C Value for ary_type
BOOLEAN	MS_BOOLEANCODE
INTEGER	MS_INTEGERCODE
LONG INTEGER	MS_LONGINTEGERCODE
REAL	MS_REALCODE
LONG REAL	MS_LONGREALCODE
BITS	MS_BITSCODE
LONG BITS	MS_LONGBITSCODE
STRING	MS_STRINGCODE
ADDRESS	MS_ADDRESSCODE
CHARADR	MS_CHARADRCODE

Arrays of pointers are not allowed.

The elements of the array should be stored as the C data type corresponding to the MAINSAIL data type of the array; i.e., they do not have to be stored as MAINSAIL or PDF data types.

A nullArray is represented by an ARRAY with ary_dims or ary_first_elem equal to zero. A uses nullArray can also be passed as null C pointer.

The array elements returned in modifies and produces array parameters are automatically allocated with malloc by RPC. The C client must deallocate the elements by calling dispose_array as described in Section 4.12.5.

4.12.2. connserver and the _init Procedure

The C procedure connserver, provided in "mrpc1.o", has the following interface:

```
int connserver(hostname,servername)
char *hostname,*servername;
```

The parameters hostname and servername are null-terminated strings that are the name of a host and the name of a service available on that host. connserver returns an integer file descriptor. If the descriptor is nonnegative, it must be passed to the appropriate _init procedure to initialize a connection with the specified service; if negative, an error occurred and no connection is established.

The _init procedure for a remote module has the name of the remote module (in lower case) with "_init" added to it, e.g., "foomod_init" for a remote module FOOMOD. The interface of an _init procedure is as follows:

```
MRPCMOD *foomod_init(fd,cliver,oldestsrvver,msg)
int fd;
long cliver, oldestsrvver;
char *msg;
```

fd is the file descriptor returned from a successful connserver call. If the _init procedure is successful, it returns a non-zero value (the "MRPCMOD *") which must be specified as the first parameter to all remote procedure calls to FOOMOD. If the _init procedure fails, it returns a zero value. msg is set to an error message if a failure occurs. cliver and oldestsrvver correspond to the cliVersion and oldestSrvVersion parameters to $newRemoteModule; see Section 4.2.

4.12.3. Calling Remote Procedures

The name of a remote procedure in C is the module name of the remote module followed by an underscore followed by the MAINSAIL name of the remote procedure, all converted to lower case; e.g., to call myProc in FOOMOD, the C procedure name would be "foomod_myproc". Remote procedures return the return value of the remote procedure; rules for C data types of return values are the same as for produces parameters. The first argument to the remote

procedure must be the value returned by the remote module's _init procedure; the subsequent arguments correspond to the MAINSAIL procedure's arguments.

4.12.4. The _final Procedure and close

To close the connection to the remote module, the C program calls a procedure with the name of the remote module (in lower case) with "_final" added to it, e.g., "foomod_final" for a remote module FOOMOD. The argument of the final procedure is the value returned from the _init procedure.

4.12.5. dispose_array

The procedure dispose_array can be used to deallocate an array (by calling cfree) if the appropriate bits are set in the ARRAY struct's ary_alloc_status field. The bits and their meanings are:

Bit	Meaning
ARY_DSCR_DYN	Free the ARRAY struct itself
ARY_DATA_DYN	Free the array elements at ary_first_elem
ARY_STRING_DYN	If the array type is string, free the text of the strings

dispose_array is declared as follows:

```
dispose_array(a)
ARRAY *a;
```

4.12.6. Handling Exceptions in the Remote Module

The C setjmp/longjmp mechanism is used to allow a C client to handle MAINSAIL exceptions in the remote module. A jmp_buf value, as returned from setjmp, is passed to rpc_register_jump along with the text of the exception that is to be handled. rpc_register_jump is declared as:

```
void rpc_register_jump(s,env)
char *s;
jmp_buf env;
```

A longjmp is made to env when the exception occurs. When the exception handling terminates, rpc_clear_jump must be called to prevent the longjmp from being taken:

```
rpc_clear_jump(s)
char *s;
```

More than one exception may be handled by the same environment (i.e., the same jmp_buf may be passed to rpc_register_jump several times to handle several different exceptions in the same place). The current exception is given by the char pointer ms_exception.

A sample use of rpc_register_jump and rpc_clear_jump appears in Example 4.12.6-1.

```
if (! setjmp(my_env)) {   /* equivalent of "$HANDLE" */
    rpc_register_jump(exception_name,my_env);
    < c code normally executed >
    rpc_clear_jump(s);
    }
else {
    /* equivalent of "$WITH", except only possible
       ultimate action is to fall out */
    ... may want to examine ms_exception ...
    ... handler code ...
    }
```

Example 4.12.6-1. Use of rpc_register_jump and rpc_clear_jump

4.12.7. Sample C RPC Session

The C program of Example 4.12.7-2 is a client that calls the interface procedure provided by the remote module Example 4.5-1; it is the C equivalent of the module of Example 4.6-2. The C program is compiled with the command shown in Table 4.12.7-3 on a typical UNIX system. The files "mrpc.h" and "mrpc1.o" have been copied to the current directory from the MAINSAIL directory; alternatively, they could be copied to a well-known (site-dependent) directory. The files "foocli.c" and "foocli.h" were produced by the C RPC compilation shown in Example 4.12.7-1, in which "foomod.msl" contains the remote module FOOMOD of Example 4.5-1.

Appendix D shows a C client that calls remote procedures with a greater range of parameter types to demonstrate the correspondence between C client arguments and MAINSAIL server parameters.

```
MAINSAIL (R) Compiler
Copyright (c) 1984, 1985, 1986, 1987, 1988, and 1989 by
 XIDAK, Inc., Menlo Park, California, USA.

compile (? for help): foomod.msl,<eol>
> rpc c<eol>
> <eol>
Opening intmod for $SYS...

foomod.msl
Opening intmod for RPCHDR...
...
Opening intmod for $SYS...
Client for FOOMOD stored on foocli.c
Header for FOOMOD stored on foocli.h
Intmod for FOOMOD not stored

compile (? for help):
```

Example 4.12.7-1. Compilation of Remote Module with "RPC C" Compile Subcommand

```
% cc -o foo foo.c foocli.c mrpcl.o<eol>
```

Table 4.12.7-3. Compiling the C Client on a Typical UNIX System

```
#include "mrpc.h"
#include "foocli.h"

MRPCMOD *rem;

main (argc,argv)
int argc;
char *argv[];
{
int fd;
char host[100],serv[100],emsg[100];
char *hostS; char *servS; char *s;

if (argc <= 2) {
    printf("Host: "); gets(host); hostS = host;
    printf("Server: "); gets(serv); servS = serv; }
else { hostS = argv[1]; servS = argv[2]; }

if ((fd = connserver(hostS,servS)) < 0) exit(1);

if (! (rem = foomod_init(fd,4,0,emsg))) {
    printf("Could not init: %s\n",emsg);
    exit();
    }

s = foomod_hello(rem,"xxx");
...
cfree(s);

foomod_final(rem); close(fd);
}
```

Example 4.12.7-2. C RPC Client in "foo.c"

5. RPC Files

RPC files provide support for file I/O operations in which the contents of the file are provided by a remote module. Application clients pass a file name and a pointer to a remote module of class $rpcFileModuleCls (see Figure 5.2-1) to the procedure $openRpcFile (see Figure 5.1-1). If the call is successful, an open file is returned upon which normal MAINSAIL I/O can be performed. The remote module is responsible for maintaining the actual contents of the RPC file.

Thus, RPC files make it possible for a server to provide "logical files" to its clients, the contents of which can be computed in arbitrary ways. For example, a server could maintain a single physical file containing a number of structure images. Clients could access each image as a separate RPC file. The server would be able to enforce file locking between the clients and to permit client access to only limited portions of the physical file.

If simple access to a system-dependent file is desired on a remote system, the NET device module (see Appendix F) provides a more transparent and convenient method than RPC files.

5.1. $openRpcFile

```
BOOLEAN
PROCEDURE $openRpcFile (
    PRODUCES POINTER(textFile) f;
    STRING fileName;
    BITS openBits;
    POINTER($rpcFileModuleCls) p;
    PRODUCES OPTIONAL STRING errorMsg);
```

Figure 5.1-1. $openRpcFile (continued)

```
BOOLEAN
PROCEDURE $openRpcFile (
    PRODUCES POINTER(dataFile) f;
    STRING fileName;
    BITS openBits;
    POINTER($rpcFileModuleCls) p;
    PRODUCES OPTIONAL STRING errorMsg);
```

Figure 5.1-1. $openRpcFile (end)

$openRpcFile opens an RPC file. The interpretation of fileName is completely up to the remote module p.

The openBits can be any combination of input, output, create, random, keepNul, delete, alterOK, $pdf, and errorOK. The detailed interpretation of all but keepNul and errorOK is up to the remote module p.

If the file can be opened, a pointer to an open file is produced, and true is returned.

If the file cannot be opened, and errorOK is set, then an error message is produced in errorMsg and false is returned.

If the file cannot be opened, and errorOK is not set, then a fatal error is signalled with errMsg. This differs from a normal MAINSAIL open since the file names passed to $openRpcFile typically have meaning only to the remote module, not to an interactive user.

RPC files support all of the MAINSAIL file system procedures that actually perform I/O, and are closed with the normal MAINSAIL close procedure. At present, RPC files do not support MAINSAIL file system procedures that do not actually perform I/O, e.g., $fileInfo.

5.2. $rpcFileModuleCls

```
TEMPORARY FEATURE:  SUBJECT TO CHANGE
```

Modules of the class $rpcFileModuleCls are RPC modules that supply the contents of RPC files. The procedures in RPC file modules are actually called by a device module written by

XIDAK. This device module is used by the files returned by calls to $openRpcFile. Thus, normal application client code never calls the procedures in an RPC file module directly.

$rpcFileOpen is called whenever a client attempts to open an RPC file by calling $openRpcFile.

The fileName supplied to the $openRpcFile call is passed without modification to $rpcFileOpen.

$openRpcFile maps a subset of its openBits into those shown in Figure 5.2-2, and passes them to $rpcFileOpen. Other bits contained in $openRpcFile's openBits are not passed to $rpcFileOpen. This mapping is necessary since the MAINSAIL-defined bits passed to $openRpcFile may change across MAINSAIL versions, but the mapped bits are guaranteed to remain the same. This makes it possible for a client and a server to be running different MAINSAIL versions.

In addition to the bits shown in Figure 5.2-2, the $rpcFileDataBit is set if the file is a data file, or the $rpcFileTextBit is set if the file is a text file. If the file is a PDF file, whether text or data, the $rpcFilePdfBit is set.

The actual interpretation of the openBits passed to $rpcFileOpen is completely up to the RPC file module. The remote module may ignore any bits, but should give an error return if passed a bit it ignores. It is expected that if an RPC file module supports one of these bits that the semantics will parallel that for the bit when passed to the normal MAINSAIL open procedure.

$openRpcFile openBits	$rpcFileOpen openBits
input	$rpcFileInputBit
output	$rpcFileOutputBit
create	$rpcFileCreateBit
random	$rpcFileRandomBit
delete	$rpcFileDeleteBit
$pdf	$rpcFilePdfBit
alterOK	$rpcFileAlterOKBit

Figure 5.2-2. RPC File openBits

bufferSize is initialized by the device module to to a positive value that is the size of buffer the device module prefers to use. For example, under SunOS TCP/IP this might be 16K, since this is the size for optimum performance. $rpcFileOpen modifies bufferSize to be the actual buffer

```
CLASS($remoteModuleCls) $rpcFileModuleCls (

    BOOLEAN
    PROCEDURE $rpcFileOpen (
        PRODUCES LONG INTEGER fileDscr;
        STRING fileName;
        MODIFIES LONG INTEGER bufferSize;
        PRODUCES LONG INTEGER eofPos;
        LONG BITS openBits;
        PRODUCES STRING errorMsg);

    BOOLEAN
    PROCEDURE $rpcFileClose (
        LONG INTEGER fileDscr;
        LONG BITS closeBits;
        PRODUCES STRING errorMsg);

    BOOLEAN
    PROCEDURE $rpcFileWriteBuffer (
        LONG INTEGER fileDscr;
        LONG INTEGER bufferNumber;
        CHARADR bufferPDF;
        LONG INTEGER bufferLength;
        PRODUCES STRING errorMsg);

    BOOLEAN
    PROCEDURE $rpcFileReadBuffer (
        LONG INTEGER fileDscr;
        LONG INTEGER bufferNumber;
        CHARADR bufferPDF;
        LONG INTEGER bufferSize;
        PRODUCES LONG INTEGER bufferLength;
        PRODUCES STRING errorMsg);
```

Figure 5.2-1. CLASS $rpcFileModuleCls (continued)

```
BOOLEAN
PROCEDURE $rpcFileWriteThenReadBuffer (
    LONG INTEGER fileDscr;
    LONG INTEGER writeBufferNumber;
    LONG INTEGER readBufferNumber;
    CHARADR bufferPDF;
    LONG INTEGER bufferSize;
    MODIFIES LONG INTEGER bufferLength;
    PRODUCES STRING errorMsg);

);
```

Figure 5.2-1. CLASS $rpcFileModuleCls (end)

size to be used by $rpcFileWriteBuffer, $rpcFileWriteBuffer, and $rpcFileWriteThenReadBuffer.

eofPos is produced by $rpcFileOpen as the end of file position of the file at the time is is opened if it can be computed by $rpcFileOpen. Otherwise it is produced as -1L.

If $rpcFileOpen can open the file, true is returned and a positive long integer is produced in fileDscr to identify the RPC file uniquely. The file descriptor is unique among all RPC files currently open by a given RPC file module. If the file cannot be opened, false is returned, and an error message is produced in errorMsg.

$rpcFileClose is called to close an RPC file. The only valid closeBit is $rpcFileDeleteBit, and its detailed interpretation is up to the RPC file module. After an RPC file is closed, the long integer used as its file descriptor may be reallocated by a subsequent $rpcFileOpen. If no errors occur, $rpcFileClose returns true, otherwise it produces the error message in errorMsg, and returns false.

$rpcFileWriteBuffer is called when the client wants to write a buffer to the RPC file indicated by fileDscr. bufferNumber indicates which buffer is to be written to the file (the first buffer in the file is buffer zero and contains character units 0L through bufferSize - 1L of the file). bufferPDF is the address of the buffer, and bufferLength is its size. bufferLength is normally equal to the bufferSize returned by $rpcFileOpen; it will be less than bufferSize only for the last buffer in the file, and will never be greater. If the buffer can be written to the file, true is returned. If an error occurs, an error message is produced in errorMsg, and false is returned.

$rpcFileReadBuffer is called when the client wants to read a buffer from the RPC file indicated by fileDscr. bufferNumber indicates which buffer is to be read from the file (the first buffer in the file is buffer zero). bufferPDF is the address of the buffer, and bufferLength is its size. bufferSize is equal to the bufferSize returned by $rpcFileOpen. If the buffer can be read from the file, true is returned, the buffer contents are filled, and bufferLength indicates the actual amount of data read into the buffer (which must always be less than or equal to bufferSize). If the requested buffer position is beyond the end of the file, true is returned, but bufferSize is 0L. If an error occurs, an error message is produced in errorMsg, and false is returned.

$rpcFileWriteThenReadBuffer is an optimization for random access files. It acts exactly as the sequence $rpcFileWriteBuffer followed by $rpcFileReadBuffer, but requires only one remote procedure call.

The bufferPDF parameter in $rpcFileWriteBuffer, $rpcFileReadBuffer, and $rpcWriteThenReadBuffer is a charadr so that it can address character units or storage units. The identifier ends in PDF only to suppress character set translation by RPC, not to indicate that the buffer necessarily contains PDF text or data; it may, in fact, contain machine-dependent text or data.

$rpcFileModuleCls is subject to change, most likely by the addition of procedures to provide additional file functionality (e.g., $fileInfo). It is possible that the name of the class and/or its fields may also change. Users writing modules of $rpcFileModuleCls must be prepared to track the applicable changes.

6. Interprocess Communication: Socket Streams

Socket streams are used for two-way interprocess communication. They do not have semantics normally associated with TTY streams: no echo occurs, there are no special keyboard interrupt characters, and there is no explicit flow control or baud rate.

The communicating processes must be aware that they are communicating with another process and they must agree on a protocol for exchange of information, including who speaks first, how the conversation is terminated, what kind of buffering is used, the required size of the buffers, what kind of objects are written (e.g., characters or storage units), and what those objects mean. This communications protocol is distinct from and "built on top of" the underlying network protocol used to carry the bytes between the processes.

Creation of sockets for client-server communication is described in Section 6.3. Creation of child processes is described in Section 6.4. Creation of sockets for communication between an arbitrary pair of processes is described in Section 6.8.

A high-level facility for remote module creation that does not require the user to deal explicitly with interprocess communication is described in Chapter 4.

6.1. Meaning of $eos on Socket Streams

On both input and output, $eos means that process at the other end has broken (hung up) the connection.

6.2. Socket-Specific Fields of $stream

The socket-specific fields of $stream are shown in Table 6.2-1.

```
STRING  $myHostName,     # Our host's name
        $hisHostName,    # Other end's host's name
        $serverName,     # The server we are talking to, or
                         # "" if we are not talking to a
                         # server.
        $protocolName;   # The network protocol name, or ""
                         # if the socket is a simulated
                         # internal socket.
```

Table 6.2-1. Socket-Specific Fields of the Stream Record

6.3. Servers and Clients: Network Protocol Modules and the SERVICE Stream Prefix

This section describes a low-level interface to server/client communication. Before a server can be run, an entry must be made in the service protocol table as described in Appendix A.

It should rarely be necessary for the typical programmer to write a low-level server; the RPC facilities described in Chapter 4 should be sufficiently general for most applications and are much simpler to use than the facilities described in this chapter.

6.3.1. Terminology and Conventions

A "service" is a set of functions available under a global (system- or network-wide) name; like a file name, the service name can be used by any process that wants to make use of the service. The service functions are available through a "high-level protocol" (defined by the author of the service) that specifies the semantics of data transferred to and from the service. The data are transmitted according to a "low-level protocol" (or "network protocol" or just "protocol") specified by the operating system; a given service may be available only through a single low-level protocol or through several different low-level protocols.

A "server" is a process that provides one or more services. When the process wishes to start handling the functions of a given service, it must make calls to the operating system to inform it that is ready to do so. In MAINSAIL, this is done with the $bindService and $acceptClient calls.

A "port" is a handle (used on the server end) that represents an active service. It is represented by a record of the MAINSAIL class $port. In a process that wishes to become a server for a particular service, the normal sequence is to verify that it is legal to become a server for that service by calling $bindService, and then, if $bindService returned a non-Zero port, to pass the port to $acceptClient. $acceptClient is typically called in a loop; each time it produces a stream, a new coroutine is created to talk through the stream.

A "client" is a process that makes use of a service. It communicates with the service through a stream. It can use the stream prefix "SERVICE" to request a connection to the service on any available low-level protocol, or it can use a stream module named for a low-level protocol to specify the low-level protocol (this may be useful where a service supports more than one low-level protocol).

A "connection" is the two-way channel between a client and a service. It is represented as a stream on each end; the server stream is obtained from the call to $acceptClient. If a server

expects to receive more than one connection at a time, it must set up a scheduled coroutine for each possible connection.

6.3.2. Service Protocol Table

A global table of registered services, the service protocol table, is maintained on each node. All services must be registered in this table before they can be used. The table format is described in Section A.1.

6.3.3. The Client End

```
$openStream(st,"service>serviceName");

        or:

$openStream(st,"service(host)>serviceName");
```

Table 6.3.3-1. Client Access to a Service

Clients may open a stream to a service using the stream prefix SERVICE. At a minimum, the service name must be provided. The host name may also be given. If the host name is omitted, $openStream automatically determines the host on which the service is available. SERVICE automatically maps to an appropriate network protocol module, e.g., TCP or VMSMBX. The available network protocol modules are listed in Appendix B.

Examples:

- "service(m)>ucrsrv": Connect to the service "ucrsrv" on host "m" using the best available common network protocol.

- "service>ucrsrv": Connect to the service "ucrsrv" on the default host using the best available common network protocol.

The network protocol module to use to talk to the service may be given instead of using the generic SERVICE stream module, although it is rarely necessary to be so specific:

- "tcp(m)>ucrsrv": Connect to the service "ucrsrv" on host "m" using the TCP network protocol.

- "vmsmbx>ucrsrv": Connect to the service "ucrsrv" on the default host using the VAX/VMS mailbox network protocol.

```
NOTE: As of this writing, when host names are specified,
      the implementation requires the syntax:

      service>host:serviceName
      tcp>host:serviceName
      vmsmbx>host:serviceName
```

6.3.4. The Server End

```
STRING
PROCEDURE    $myHostName;

STRING
PROCEDURE    $canonicalHostName
                          (STRING hostName);

INTEGER
PROCEDURE    $getHosts    (STRING serviceName;
                           MODIFIES STRING ARRAY(1 TO *)
                               hosts;
                           PRODUCES OPTIONAL STRING eMsg);

INTEGER
PROCEDURE    $getProtocols
                          (STRING hostAndServiceName;
                           MODIFIES STRING ARRAY(1 TO *)
                               prots;
                           PRODUCES OPTIONAL STRING eMsg);
```

Table 6.3.4-1. Server Procedures (continued)

```
LONG INTEGER
PROCEDURE    $unbindService
                          (MODIFIES POINTER($port) p;
                           OPTIONAL LONG INTEGER timeout;
                           OPTIONAL BITS ctrlBits);

LONG INTEGER
PROCEDURE    $acceptClient
                          (POINTER($port) p;
                           PRODUCES POINTER($stream) st;
                           OPTIONAL LONG INTEGER timeout;
                           OPTIONAL BITS ctrlBits);

BOOLEAN
PROCEDURE    $parseHostServiceName
                          (STRING name;
                           PRODUCES STRING host,service;
                           PRODUCES OPTIONAL STRING eMsg);
```

Table 6.3.4-1. Server Procedures (end)

A server typically uses $getProtocols to discover the network protocols it is expected to support for each service. It then starts a separate coroutine to handle each protocol/service pair. For each such pair, the server uses $bindService to bind itself to a port for the protocol and service and repeatedly calls $acceptClient to accept new clients on that port. Whenever $acceptClient returns a new stream, a new coroutine is created to talk through that stream and perform the service functions on the connection represented by the stream.

$myHostName returns the host name of the current host. If the host name is not available, $myHostName returns the null string. The $myHostName field of a socket stream st gives the host name of the system on which the program is running, i.e.:

```
st.$myHostName = $myHostName
```

$canonicalHostName returns the official name of the host named by hostName according to the MAINSAIL services table. It returns the original string if hostName is not in the table.

$getHosts returns the host names that support the service serviceName. In this way, a server may determine if any other copies of itself might be running on other systems. $getHosts is also used internally when a client opens a stream with a name of the form "service>xxx"

(where xxx is a service). In this case, if the service xxx is uniquely available on a single host, the host name is filled in automatically. $getHosts returns 0 with eMsg set to an appropriate error message if an error occurs.

$getProtocols returns the network protocols through which hostAndServiceName may be accessed from the current host. The call is provided for servers so they can set themselves up to service the appropriate set of network protocols.

$getProtocols modifies the input array (allocating it if nullArray is passed) to contain all of the nework protocols required to support the specified service on the specified host, and it returns the number of such protocols in the array. The information returned by $getProtocols is currently derived from the service protocol table. If no protocols are available, $getProtocols returns 0, with eMsg set to an appropriate error message. hostAndServiceName may take one of the forms shown in Table 6.3.4-2.

hostAndServiceName	Returned Network Protocols
""	All that the current host speaks
"service"	All under which "service" is available on the current host
"(host)>service"	All under which "service" is available on the host with the given name

NOTE 1: Both the host and service name are assumed to be valid MAINSAIL identifiers.

NOTE 2: As of this writing, the only accepted syntax is still "service>host:serviceName"

Table 6.3.4-2. Valid hostAndServiceName values ($getProtocols)

$bindService accepts a single network protocol and service name and returns a pointer to a port. A server typically binds itself to one port for each network protocol that the host system supports.

The protocolAndServiceName argument has the format:

```
protocol(host)>service
```

or, in the current release:

```
protocol>host:service
```

The "protocol>" and "host:" parts are optional. If the host is omitted, the current host is the default. If the protocol is omitted, the protocol defaults to the first one listed in the service protocol table for the named service on the given host. The available protocol modules are listed in Appendix B.

Table 6.3.4-3 shows the user-visible fields of a $port record. The service name and host name are the parsed service and host name supplied to the $bindService call, or if they were not supplied, to the values to which they defaulted. $lastError is set by $acceptClient if an error return is made.

```
CLASS $port (
    STRING
        $serviceName,   # As supplied to $bindService
        $hostName;      # "    "        "    "
    STRING
        $lastError;     # Set by $acceptClient
);
```

Table 6.3.4-3. Fields of $port

If the service cannot be bound, $bindService returns nullPointer with eMsg set to an appropriate error message. If timeout is a positive number, the call returns when that many milliseconds have elapsed. Specify the value $block to wait indefinitely.

The $unbindService procedure is used if the server decides to stop accepting clients on a port.

The $acceptClient procedure accepts a port (returned by $bindService) and produces a stream. It blocks until a client requests a connection to the server on the port p (the client uses $openStream to open its end). The server then talks to the client through the stream produced by $acceptClient. At the end of the interaction, the server and the client each call $closeStream on their respective streams to terminate the connection. $acceptClient returns standard stream error codes; i.e., "$success($bindService(...))" returns true if the service can be bound, otherwise false, with p.$lastError set to an error message string. On a false return, the value is

either $error or $timedOut, if a positive timeout value (milliseconds) was given. Specify $block for timeout if the procedure should wait indefinitely.

$parseHostServiceName parses a "(host)>service" name into the strings host and service, handling quoted names and the fact that the host name may be omitted. This procedure is rarely needed by user code. It returns false with eMsg set to an explanation if it cannot parse the name.

6.3.5. The Service Protocol Table

The server/client implementation currently depends on the availability of a file describing the services available on each host. The logical name "(service protocol table)" must be defined using a MAINEX "ENTER" subcommand in the MAINSAIL startup command file to specify the file containing the table.

The service protocol table may eventually be replaced by a XIDAK name server. See Appendix A.

6.4. Child Process Creation: SOCPRO and PTYPRO

In the parent/child process control model, a parent process creates one or more child processes to do its bidding. The children may in turn become parents of additional processes. The child process currently runs on the same node as the parent unless the NETSTR stream module is used to access a remote node.

When the parent process exits, the children processes (and all of their children) are usually terminated if they are still active.

The parent/child process control model is similar to the scheduled coroutine model. In fact, it is possible to write programs that communicate through a stream with some other entity that is either a separate process or another coroutine in the current process.

6.4.1. Starting a Child Process Using SOCPRO

Child processes are started using the $openStream procedure and the stream module SOCPRO. The name of the stream has the form:

```
"socpro>programName programArgs <eol>
 userName <eol>
 password <eol>
 directoryName"
```

or:

```
"socpro(hostName)>programName programArgs <eol>
 userName <eol>
 password <eol>
 directoryName"
```

In the first case, a new process is started on the current host by running the program stored in the file named by programName. If a host name is specified, as in the second case, the new process is started on that host instead of the current system.

If programArgs is given, those arguments are made available to the program, provided that program arguments are supported by the operating system. The process is started with the same user ID as the current process, unless userName is given, in which case it is used to set the user ID for the process. The process's current directory is the same as the parent process's (if on the same machine) or the home directory for the user ID (if on a remote machine).

Password is required if the current process would not normally have the right to run as the specified userName.

The $openStream call returns a socket stream that talks directly to the child's $tty stream. Thus, for the child, "$isatty($tty)" returns false. Input for the child may be supplied by calling $writeStream, and output from the child may be read using $readStream.

Characters written to the process stream are not echoed back. This form of process control is useful for direct interprocess communication. It is very similar to the use of pipes on UNIX.

6.4.2. Starting a Child Process Using PTYPRO

A child process may also be started using the $openStream procedure and the stream module PTYPRO. The name of the stream has the form:

```
"ptypro>programName programArgs <eol>
 userName <eol>
 password <eol>
 directoryName"
```

or:

```
"ptypro(hostName)>programName programArgs <eol>
 userName <eol>
 password <eol>
 directoryName"
```

A new process is started on the current host or on the specified host by running the program stored in the file named programName. If neither programName nor programArgs is given, and the system supports running a "shell" as a child process, the standard system shell is run. The remaining arguments are optional and are interpreted as for SOCPRO.

The $openStream call returns a PTY stream to talk to the child's $tty. Thus, for the child, "$isatty($tty)" is true. Talking to the PTY is much like sitting at a keyboard in front of a terminal screen. Characters written to the stream are echoed back, if the running program wants them to be.

This kind of connection has a specialized but useful application for system shells and windowing systems. The program that creates the process (e.g., the shell or window manager) serves merely as the intermediary between the actual user and the created process.

Special considerations of the PTY stream type are discussed in Section 10.9.

Many systems do not provide general pseudo-terminal support. On such systems, PTYPRO is not available.

6.4.3. Starting a MAINSAIL Child Process

```
STRING
PROCEDURE    $executableBootName
                          (OPTIONAL STRING hostName);
```

Table 6.4.3-1. $executableBootName

$executableBootName returns the name of the standard MAINSAIL executable bootstrap on hostName (on the local system if hostName is omitted). It returns the null string if the information is not available.

A MAINSAIL child process can be started on the current system with a stream name of the form:

```
"socpro>" & $executableBootName
```

If the operating system supports command line arguments, a particular MAINSAIL module can be started in a child process using a stream name of the form:

```
"socpro>" & $executableBootName & " " & modName
```

where modName is the desired MAINSAIL module.

IMPLEMENTATION NOTE

$executableBootName is currently implemented using a logical file name. $executableBootName calls lookupLogicalName with the argument:

```
"(executable boot " & hostName &")"
```

For this call to succeed on systems other than the current host, the installer of MAINSAIL must define the standard bootstrap names of all host systems in MAINEX "ENTER" subcommands (typically in the local "site.cmd" file). The implementation may be changed in the future to use a more flexible mechanism.

6.4.4. Establishing an Additional Control Stream to a Cooperating Child

A cooperating child process is a program designed to talk to a parent, much as a server process is designed to talk to a client. It is written to use a predefined communications protocol for exchange of information with its parent.

Typically, the cooperative communication with the child takes place over a control stream that is separate from the child's $tty. This control stream is available to the child as the STREAMS system variable $parent and may be established by the parent using the two-pointer form of $openStream, e.g.:

```
$openStream(childTty,childControl,"socpro>.....");
```

Typically, childTty is used by the child only for error messages and when the debugger is invoked in the child. The childControl stream is typically used for all cooperative communication between parent and child.

6.5. Terminating a Child Process

A child process created with $openStream may be terminated by closing the stream with $closeStream. If the process terminates by itself prior to the $closeStream call, the currently active or next I/O call to the process's stream returns with an error.

6.6. Process Control

If the started process is a cooperating child program designed to run as a child process, the parent and child can use a communications protocol of their choice. Any stream I/O mechanism may be used, including $packet I/O.

If the started process is a non-cooperating child program connected to a PTY stream, a special character may be sent to the process to cause it to interrupt. This character is provided as a field of the class $stream (see Chapter 10). Alternatively, the procedure $writeStreamInterrupt may be used to interrupt a non-cooperating child process. If the child process has enabled interrupt handling, it may do something special based on the interrupt. Otherwise it will probably die and further reads and writes to its stream will return an error.

No other process control is currently provided.

6.7. Examples

6.7.1. Sprouting a Print Job and Waiting for It to Exit

The most simple interaction with a child process is running a non-interactive program and waiting for it to complete. This form of child process control does not require scheduling. Example 6.7.1-1 is an example of invoking a print program directly from a MAINSAIL program.

```
PROCEDURE printFile (STRING name);
BEGIN
STRING s;
POINTER($stream) child;

$openStream(child,"socpro>print " & name);
WHILE $success($readStream(child,s,errorOK)) DO
    write(logFile,s);
$closeStream(child);
END;
```

Example 6.7.1-1. Sprouting a Print Job and Waiting

printFile uses a program called "print" to print a file given its name. Any terminal output generated by the print program is written to its parent process's logFile. When the print program terminates, $readStream returns an end-of-stream and the child stream is closed.

This simple example does not require scheduling of advanced STREAMS for the system on which it is run. Note, however, that no provision for providing input to the child is provided. If the child program attempts to read from its terminal, it blocks until some input is supplied. The parent is also blocked in $readStream waiting for the child to output or terminate, so both processes may block indefinitely.

6.7.2. Sprouting an Interactive Child

The more general case of handling an interactive child process, that is, a process that performs terminal input, is shown in Example 6.7.2-1. The child is started much the way a program is stared by a system shell. All of the child's output goes to the parent's terminal, and input from the parent's terminal goes to the child. This form of child process control requires scheduling.

runCmd runs a child program. The child's input and output are connected to the stream returned by $openStream, so the parent must handle both using two independent coroutines. reader is started as a coroutine to read what the child writes and output it to the parent's $tty. writer is started as a coroutine to read from the parent's $tty and write the keystrokes to the child process.

The parent might also perform processing of the I/O to the child, such as logging the child's output or supplying its input from a file. Also, typically the parent will want to enable and handle keyboard interrupts and pass them on to the child. Neither of these possibilities is shown.

If either the reader or writer detects the end of the child process, both the reader and writer are killed and runCmd returns to its caller in the parent.

```
POINTER($stream) pty;
POINTER($coroutine) rc, wc;

DEFINE print(x) = [$writeStream($tty,x,$line)];
```

Example 6.7.2-1. Sprouting an Interactive Child (continued)

```
PROCEDURE reader;
BEGIN
STRING s;
WHILE $success($readStream(pty,s,errorOK)) DO print(s);
IF wc AND NOT $killedCoroutine(wc) THEN
    $killCoroutine(wc);
$reschedule(delete);
END;



PROCEDURE writer;
BEGIN
STRING s;
DO $readStream($tty,s)
    UNTIL  NOT $success($writeStream(pty,s,errorOK));
IF rc AND NOT $killedCoroutine(rc) THEN
    $killCoroutine(rc);
$reschedule(delete);
END;


PROCEDURE runCmd(STRING args);
BEGIN
STRING s, t;
OWN INTEGER nRun;
IF NOT $openStream(pty,"ptypro>"&args,errorOK,s) THENB
    printError(s); RETURN END;
```

Example 6.7.2-1. Sprouting an Interactive Child (continued)

```
$HANDLEB
    startup; nRun.+1;
    s := "RunReader" & cvs(nRun);
    t := "RunWriter" & cvs(nRun.+1);
    rc := $createCoroutine(thisDataSection,"reader",s);
    $queueCoroutine(rc);
    wc := $createCoroutine(thisDataSection,"writer",t);
    $queueCoroutine(wc);
    $waitForDescendants;
    $closeStream(pty) END
$WITHB
    IF $exceptionName = $abortProcedureExcpt THENB
        IF pty THEN $closeStream(pty); cleanup END;
    $raise END;
END;
```

Example 6.7.2-1. Sprouting an Interactive Child (end)

6.8. Process Rendezvous Communication [Not Implemented]

In the rendezvous model, a process creates a "rendezvous" stream name and passes the name (as a string) to two arbitrary processes. At some later time, the two processes may establish communication with each other by opening a rendezvous stream using the "rendezvous" stream name.

6.8.1. Creating a Rendezvous Stream Name

```
STRING
PROCEDURE   $createRendezvousName
                            (OPTIONAL BITS ctrlBits;
                             OPTIONAL STRING msg);
```

Table 6.8.1-1. $createRendezvousName

$createRendezvousName returns a new rendezvous name. The name may be used by processes to establish communication with each other. It is guaranteed to be unique to the network on which the process is running.

If the errorOK bit is given, $createRendezvousName returns the null string and sets msg when an error occurs. Otherwise, the procedure calls errMsg on errors. The most common error is the inability to access the name server on the current system.

6.8.2. Symmetric Communication Using a Rendezvous Name

Given a rendezvous name, a process can establish communication with another process using a normal $openStream call, e.g.:

```
$openStream(st,rendezvousName);
```

The $openStream procedure waits until some other process requests a rendezvous on that name. It then returns a socket stream over which further communication may occur.

The communications protocol must be known in advance by both processes. The protocol is typically either known in advance by both processes or indicated by the process that provided the rendezvous name.

6.8.3. Server/Client Communication Using a Rendezvous Name

Given a rendezvous name, one of the processes may elect to become a server by using the rendezvous name as a port name rather than as a stream name. Several other processes may then establish communication with the server using $openStream, as described in the previous section. The server process does a normal $bindService call, e.g.:

```
port := $bindService(rendezvousName);
```

and then does $acceptClient calls on the port returned by $bindService.

6.8.4. Underlying Implementation

The rendezvous name currently has the form:

```
rendezvous(rendezvousHost)>uniqueName
```

where rendezvousHost is the host name of the process that called $createRendezvousName and uniqueName is a name provided by a rendezvous service running on rendezvousHost that is unique to that server. The processes that wish to communicate call $openStream on the name, which causes a communication with the rendezvous server on rendezvousHost to find out how to locate and communicate with the other process.

7. Multitasking and the Scheduler

> PORTABILITY WARNING
>
> Not all operating systems support the ability to schedule stream I/O. Many systems, such as BSD UNIX and VAX/VMS, support scheduling on all stream types. Some operating systems do not support the ability to schedule any stream types at all, and some restrict scheduling to either TTY streams or non-TTY streams.
>
> To be portable, programs must check whether timeout is supported by the Scheduler on the current system. If they contain logic that relies on timeouts, that logic must be conditional on timeout support. Programs must also check whether TTY and advanced STREAMS support scheduling on the current system, and handle both possibilities. A summary of how to test adaptively for support of various stream facilities may be found in Chapter 2.

Using coroutines, a single MAINSAIL program may perform multiple "simultaneous" I/O tasks in parallel. The effect is similar to the use of several independent processes except that the coroutines may share data structures and the scheduling is, in general, more efficient.

To use scheduled stream I/O, a program sets up one coroutine for each independent stream I/O task that it wishes to perform. Each coroutine then performs the I/O in the usual way, as if it were the only coroutine running.

When a stream input or output procedure is called by more than one coroutine, the STREAMS package schedules the next runnable coroutine, i.e., the next coroutine that can perform its operation without blocking. The Scheduler, a special STREAMS system coroutine, makes a single operating system call that waits until the first of the pending I/O operations can be performed, and it then resumes the coroutine for which the operation is complete. Coroutines manipulated by the scheduler are called "scheduled coroutines"; scheduled coroutines are distinguished from other coroutines only in that they are manipulated by the Scheduler. Scheduled coroutines may be created with $createCoroutine like any other coroutine.

Rescheduling can occur whenever a coroutine performs I/O to a stream or when it explicitly calls a Scheduler procedure (see Tables 7.2-1, 7.3-1, 7.4-1, 7.5-1, and 7.6-1). The calling coroutine is not resumed until the reason for scheduling has been satisfied, at which point it becomes eligible to be resumed. If more than one coroutine is eligible to run, the Scheduler maintains a FIFO queue of ready coroutines and uses a round robin approach to reschedule the ready coroutines in the order they become ready.

The STREAMS Scheduler performs functions similar to a multiprocess operating system's process scheduler. It schedules stream I/O operations to achieve the appearance of asynchronously running processes. Unlike an operating system, however, it cannot interrupt running coroutines based on an allotted time quantum alone. Thus, if one coroutine gains control and then begins a CPU-intensive computation that involves no I/O, that coroutine continues to run until it performs stream I/O or voluntarily makes an explicit call to the Scheduler to allow other coroutines to run.

When a coroutine is stopped waiting for the Scheduler to resume it, the effects are undefined if any coroutine other than the Scheduler resumes it.

7.1. Cautions About Shared Data Access

Programmers using coroutines must be aware that any variables may be modified by any coroutine that happens to be running that has access to the variable. Coroutines that share a common data structure should use the locking semaphores described below to ensure that any shared data structures are updated correctly.

Another source of potential trouble is procedure re-entrancy. The same procedure may be in the course of executing in several different coroutine execution threads. Own variables in such procedures must be considered a shared global resource if there is any possiblility that the procedure might be re-entered by another coroutine before the current coroutine has exited it, due to the current coroutine being rescheduled.

Many conditions can cause rescheduling, including calls to errMsg and all I/O calls (even ordinary disk and terminal I/O). Since error conditions are difficult to predict, the use of semaphores for shared data access of questionable safety is recommended.

7.2. Scheduling Coroutines for Stream I/O: $queueCoroutine

```
BOOLEAN
PROCEDURE    $queueCoroutine
                              (POINTER($coroutine) c;
                               OPTIONAL BITS ctrlBits);
```

Table 7.2-1. Creating and Scheduling Coroutines

Coroutines that perform scheduled I/O ("scheduled coroutines") are created using $createCoroutine just like any other coroutine. Instead of using $resume to resume them, the Scheduler procedure $queueCoroutine is used to put a coroutine on the ready list, ready to run when the next rescheduling takes place.

Coroutines that do scheduled I/O are killed using $killCoroutine, just like any other coroutine.

7.3. Voluntary Rescheduling: $msTimeout and $reschedule

```
PROCEDURE    $msTimeout   (LONG INTEGER milliseconds);

PROCEDURE    $reschedule (OPTIONAL BITS ctrlBits);
```

Table 7.3-1. Voluntary Rescheduling Calls

$msTimeout blocks the calling coroutine until the specified number of milliseconds has elapsed. The timeout value must be positive.

$reschedule places the currently running coroutine at the end of the ready queue. A coroutine may wish to do this to timeshare itself when it performs long computations that do not involve stream I/O or any other calls that enter the Scheduler.

If the delete is specified in ctrlBits, the currently running coroutine is killed and the Scheduler is resumed so that other eligible coroutines may be run.

7.4. Waiting for Descendant Coroutines: $waitForDescendants

```
LONG INTEGER
PROCEDURE    $waitForDescendants
                         (OPTIONAL POINTER($coroutine)
                              ARRAY(1 TO *) children;
                          OPTIONAL LONG INTEGER timeout);

LONG INTEGER
PROCEDURE    $waitForDescendants
                         (POINTER($coroutine) child;
                          OPTIONAL LONG INTEGER timeout);
```

Table 7.4-1. Coroutine Synchronizing Calls: $waitForDescendants (Generic)

$waitForDescendants blocks until the coroutines in the children array have died. If the array is omitted or nullArray is given, it waits until all of its children have died. In the second form, the procedure waits until a single child coroutine has died.

It is an error to wait for coroutines that are not descendants of the calling coroutine.

7.5. Semaphores (Locks)

```
CLASS $semaphore (
    STRING $name;
);

POINTER($semaphore)
PROCEDURE    $newSemaphore
                         (STRING name);
```

Table 7.5-1. Semaphore Scheduler Calls (continued)

```
PROCEDURE    $disposeSemaphore
                          (MODIFIES POINTER($semaphore)
                              sem);

LONG INTEGER
PROCEDURE    $lock        (POINTER($semaphore) sem;
                           OPTIONAL BITS ctrlBits;
                           OPTIONAL LONG INTEGER timeout);

LONG INTEGER
PROCEDURE    $unlock      (POINTER($semaphore) sem;
                           OPTIONAL BITS ctrlBits);

BOOLEAN
<macro>      $isLocked(sem);
```

Table 7.5-1. Semaphore Scheduler Calls (end)

Semaphores provide a mechanism through which a set of cooperative coroutines can share common data structures and perform manipulations on them without risking an invalid simultaneous update. The Scheduler delays the execution of a coroutine that requests a semaphore until the semaphore is available.

Using semaphores, an application can build higher-level locking mechanisms. For example, the MEMSTR stream module is implemented using semaphores, and XIDAK database software uses semaphores to implement read and write locks on database records with deadlock detection.

A semaphore is created by calling $newSemaphore. Its initial state is unlocked. A semaphore name is supplied when creating the semaphore. The name helps identify the semaphore in error messages, during debugging, and in the scheduled coroutine map. It need not be unique.

$semaphore may contain additional, undocumented fields besides $name.

The procedures $lock and $unlock lock and unlock a semaphore record. Once the semaphore is locked, any coroutine that attempts to lock it blocks until some other coroutine unlocks it. An optional timeout allows the lock request to time out if it cannot be performed within the given timeout period. Unlocking is always performed immediately. If errorOK is given in ctrlBits, error messages are suppressed for $lock and $unlock; otherwise, fatal error messages are issued.

The macro $isLocked may be used to query the state of a semaphore without rescheduling the current coroutine. It evaluates to true if the semaphore is locked and false if not.

A semaphore may be disposed by calling $disposeSemaphore. Any coroutines waiting for the semaphore get an error return from their $lock calls.

It is the responsibility of the programmer using semaphores to ensure that they are unlocked and/or disposed of properly when coroutines that need them are killed. Through injudicious use, it is possible to create deadlocks.

7.6. Scheduled Coroutine Map

```
TEMPORARY FEATURE:  SUBJECT TO CHANGE
```

```
STRING
PROCEDURE    $scheduledCoroutineMap;
```

Table 7.6-1. Scheduled Coroutine Map

$scheduledCoroutineMap returns a string showing the currently queued I/O requests by coroutines, the streams currently blocked for I/O, the state of the coroutines and streams in the map, and all locked semaphores. This procedure may be invoked interactively by invoking the module SCOMAP.

7.7. Example Multitasking Programs

7.7.1. A Simple Terminal Emulator

A standard terminal emulator (such as the one implemented in MAINKERMIT to allow the user to log into a remote system) must perform the following independent functions:

- Read characters from the keyboard as they are typed, and send them to the remote system through a serial line without echoing them on the local screen.

- Read characters from the remote serial line and display them on the screen as they are received.

These functions are shown schematically in Figure 7.7.1-1.

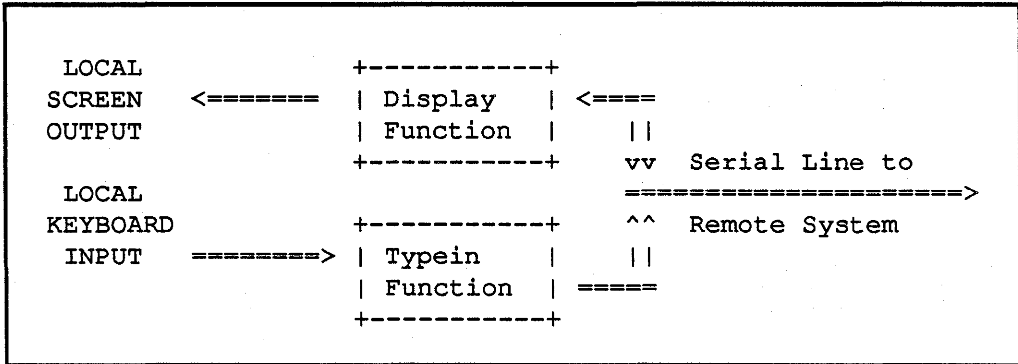


Figure 7.7.1-1. Schematic of a Terminal Emulator Program

Example 7.7.1-2 shows how these two independent functions might be implemented as two coroutines that perform stream I/O. The initial procedure creates and queues a coroutine to perform each of the functions, and then waits for both coroutines to complete. Each coroutine keeps copying until its read fails, and then the coroutine kills itself. The actual termination sequence is not shown in this simple example.

The variables $tty and serialLine refer to the controlling terminal (keyboard and screen) and an independent serial line to the remote computer, respectively.

Each coroutine is written as if it were doing normal blocking I/O. However, the Scheduler and the stream I/O system make sure that neither coroutine causes the program to go into a blocked state, thus preventing the other coroutine from running when input is available to it.

In general, a MAINSAIL program may contain an unlimited number of such coroutines. For example, to implement a windowing environment, a pair of coroutines similar to the ones above might be used for each simultaneously active window on the screen.

7.7.2. Parallel Processing Using RPC Calls

In applications that are easily divisible into several tasks, each independent of the results of the other tasks, a significant speedup can be achieved using multiple coroutines to make remote

```
BEGIN "emul8"

PROCEDURE displayFunction;
BEGIN
CHARADR buffer;
LONG INTEGER ii;
<allocate buffer>
WHILE $success(ii :=
        $readStream(serialLine,buffer,bufSize,errorOK))
    DO  $writeStream($tty,buffer,ii);
<deallocate buffer>
$reschedule(delete);
END;



PROCEDURE typeinFunction;
BEGIN
CHARADR buffer;
LONG INTEGER ii;
<allocate buffer>
WHILE $success(ii :=
        $readStream($tty,buffer,bufSize,
                    errorOK!$noFlow!$noInterrupt))
    DO  $writeStream(serialLine,buffer,ii);
<deallocate buffer>
$reschedule(delete);
END;
```

Example 7.7.1-2. A Simple Terminal Emulator Using STREAMS (continued)

```
INITIAL PROCEDURE;
BEGIN
$queueCoroutine($createCoroutine
        (thisDataSection,"displayFunction"));
$queueCoroutine($createCoroutine
        (thisDataSection,"typeinFunction"));
$waitForDescendants;
END;

END "emul8"
```

Example 7.7.1-2. A Simple Terminal Emulator Using STREAMS (end)

procedure calls to different processes in overlapping time, rather than performing the tasks sequentially in the same coroutine.

For example, the initial procedure in Example 7.7.2-1 starts up remote procedure calls to several different instances of the same remote module REMMOD in parallel and returns when all of them are done. If the different instances run on different CPU's, true parallel processing occurs.

The program sets up the array "task" to contain different arguments to be passed to the remote module remMod. It also sets up the array "host" to contain host names on which to execute each instance of REMMOD, and the array "result" to hold the answers.

The initial procedure of the program starts "task.ub1" coroutines executing the doCall procedure, one for each argument. It then waits for all of the coroutines to finish (die).

Each instance of doCall starts a child process running a unique instance of REMMOD, passes it the argument, and receives back the result (the argument is a modifies parameter). The doCall coroutine kills itself when it has the answer (the answer has been saved in a global array).

The example above has been kept fairly simple for clarity. It could be enhanced in several ways:

- doCall could call more than one remote procedure in REMMOD.
- The program could handle an unexpected termination of child processes and restart them, possibly on a different node.

```
BEGIN "ll"

RESTOREFROM "rpchdr";

CLASS($remoteModuleCls) remModCls
    STRING PROCEDURE proc (STRING task);
);

INTEGER argNum;
STRING ARRAY(1 TO *) task,result,host;

PROCEDURE reader;
BEGIN
STRING s;
POINTER($stream) ctty;
ctty := $thisCoroutine.$userHook; # Copy my TTY
$thisCoroutine.$userHook := NULLPOINTER;
WHILE $success($readStream(ctty,s,errorOK)) DO
    $writeStream($tty,s,$line);
$reschedule(delete);
END;



PROCEDURE doCall;
BEGIN
INTEGER i;
POINTER(remModCls) m;
POINTER($stream) chCtl,chTty;
POINTER($coroutine) readerCo;

i := argNum;

# Create the child
$openStream(chTty,chCtl,
    "socpro(" & host[i] & ")>" &
        $executableBootName & " rpcsrv - remMod");
```

Example 7.7.2-1. Multitasking RPC Calls (continued)

```
# Handle its TTY output, in case of errors
readerCo := $createCoroutine(thisDataSection,"reader");
readerCo.$userHook := chTty; # Pass stream as parameter
$queueCoroutine(readerCo);

# Use it to supply remMod
m := $newRemoteModule(chCtl,"remMod");
result[i] := m.proc(task[i]);
...
dispose(m);

# Kill the child
$closeStream(chTty); # kills the reader implicitly
$closeStream(chCtl);

# Kill this coroutine
$reschedule(delete);
END;



INITIAL PROCEDURE;
BEGIN
INTEGER i;
... # set up the arrays task, host, and result
# Start task.ub1 coroutines and wait for them to finish
FOR i := 1 UPTO task.ub1 DOB
    globalI := i;
    $queueCoroutine($createCoroutine
                    (thisDataSection,"doCall")) END;
$waitForDescendants;
END;

END "11"
```

Example 7.7.2-1. Multitasking RPC Calls (end)

- A dynamic mechanism could be used to select hosts on which to execute each subtask, and the program could be modified to work with fewer hosts than there are subtasks.

Note that each coroutine creates a different instance of the remote module. Calling the same instance of a remote module in different coroutines does not speed up the program, because all of the calls to a single instance of a remote procedure are processed by the same process and go through the same communication channel.

8. Handling Keyboard Interrupts

The STREAMS package provides the ability to detect interrupts generated on the physical $tty in a portable way, provided that this capability is supported by the underlying operating system. The interrupt is typically generated by typing some control character such as CTRL-C on the keyboard of the program's controlling terminal. Using an exception handler, a portable MAINSAIL program can handle such interrupts in what appears to be an asynchronous way.

8.1. Enabling and Handling Interrupts on $tty

```
BOOLEAN
PROCEDURE   $enableInterrupts
                        (OPTIONAL INTEGER
                             exitAfterThisMany);

PROCEDURE   $disableInterrupts;

INTEGER
PROCEDURE   $interruptsEnabled;

# Predefined identifier for the interrupt governed by the
# above procedures:

# system variable
STRING $keyboardInterruptExcpt;
```

Table 8.1-1. Interrupt Catching Procedures

Currently, these procedures work only on the physical TTY stream $tty (see Section 10.1) associated with a process. If multiwindow support is implemented, these procedures may be extended to work on simulated TTY streams that refer to a window.

By default, keyboard interrupt catching through STREAMS is disabled. This means that the interrupt character typically either aborts the program or confirms the interrupt and then aborts the program.

When the procedure $enableInterrupts is called and the system supports catching interrupts on $tty, the procedure returns true; interrupt catching through STREAMS is then enabled. If interrupt catching on $tty is not supported by the system, the procedure returns false.

Once interrupt catching is enabled, the coroutine that most recently called $enableInterrupts (or possibly one of that coroutine's ancestors) may catch the exception $keyboardInterruptExcpt using a MAINSAIL exception handler. The exception is raised by the Scheduler.

exitAfterThisMany may be supplied to provide an "escape hatch" in case the Scheduler does not receive control for some reason (the program might be in a tight loop that does not involve scheduled I/O). The exitAfterThisMany value applies only to the physical terminal, not to (possible future) simulated $tty's, since the Scheduler will always be involved when generating a simulated interrupt.

If exitAfterThisMany physical keyboard interrupts occur without a rescheduling, the interrupt is processed in the default way as if the program had not issued the $enableInterrupts call (typically the program is aborted). If exitAfterThisMany is not given or is 0, 10 is used by default. If exitAfterThisMany is -1, the escape hatch feature is disabled. Programs that do $enableInterrupts(-1) must be very carefully written so they never go into an infinite loop.

$disableInterrupts restores the previous behavior of keyboard interrupt handling. Calls to $enableInterrupts and $disableInterrupts may be nested, in which case $disableInterrupts restores the previous caller's parameters.

It is the programmer's responsibility to ensure that the exception $keyboardInterruptExcpt is actually handled. It is also the programmer's responsibility to ensure that a matching $disableInterrupts occurs for each $enableInterrupts. In particular, be sure to call $disableInterrupts if the $abortProcedureExcpt occurs in the handling coroutine.

$interruptsEnabled returns the non-zero value of exitAfterThisMany, or zero if interrupts are not currently enabled by the calling coroutine.

Example 8.1-2 shows the recommended form of a keyboard interrupt handler. Interrupt handling is enabled and disabled within the handle statement, to guarantee that there is always a handler for the keyboard interrupts. Interrupt handling is also disabled when $abortProcedureExcpt is raised, to clean up correctly in case the current procedure is aborted (e.g., due to an unanticipated error).

8.2. Stacking of $enableInterrupts

Enabling may be dynamically stacked, that is, a procedure that was called by the first enabling procedure may also enable interrupts. Enabling may also be stacked with respect to coroutines;

```
$HANDLEB
    $enableInterrupts;
    ...
    $disableInterrupts END
$WITHB
    IF $exceptionName = $keyboardInterruptExcpt THENB
        IF ... THEN $raiseReturn
        EL disableInterrupts; # and fall out of handler
        END
    EF $exceptionName = $abortProcedureExcpt THENB
        $disableInterrupts; $raise END
    EL  $raise END;
```

Example 8.1-2. Handling Keyboard Interrupts

i.e., a coroutine that is either a descendant or an ancestor of all enabled coroutines may also enable. No other coroutines may enable; it must be the case that there is a direct line of coroutine ancestry through all enablers. An error message is issued if this rule is violated.

The coroutine of the most recent enabler receives the exception. If it does not handle $keyboardInterruptExcpt by falling out of the handler or by doing a $raiseReturn, the exception is propagated to the ancestor coroutines in the normal way. It is an error if the exception is not handled at all by any coroutines. In this case, the Scheduler disables the most recent enabler, and possibly its ancestors as well.

If all enablers disable, handling of interrupts is disabled, that is, the default action for keyboard interrupts (typically aborting the program) is restored.

8.3. Conflict between $enableInterrupt and the $noInterrupt Bit

The $noInterrupt bit in a read, if implemented, takes precedence over the state of enabled interrupts. Thus, if a program is set up to catch interrupts on $tty and a procedure running in any coroutine calls "$readStream($tty,...,$noInterrupt...)" on the same $tty, the interrupt characters are read as normal characters by $readStream. Interrupts remain turned off until an input from $tty occurs in which $noInterrupt is not given.

8.4. Sample Program That Catches Interrupts

Example 8.4-1 shows an interactive program (command interpreter) that catches interrupts.

```
PROCEDURE commandInterpreter;
BEGIN
STRING cmd;
DOB write(logFile,"Command: "); read(cmdFile,cmd);
    IF equ(cmd,"exit",upperCase) THEN RETURN
    EF isValidCmd(cmd) THENB ... END
    EL errMsg("Invalid command","",noResponse) END;
END;



PROCEDURE topLevel;
BEGIN
$enableInterrupts;
DOB $HANDLEB commandInterpreter; DONE END
    $WITHB
        IF $exceptionName = $keyboardInterruptExcpt THEN
            # fall out
        EF $exceptionName = $abortProcedureExcpt THENB
            $disableInterrupts; $raise END
        EL  $raise END END;
$disableInterrupts;
END;
```

Example 8.4-1. Catching Interrupts Asynchronously

This approach is similar to that taken by a system shell. The top-level procedure, topLevel, calls a command interpreter that read commands in a loop. If an interrupt occurs, the handler in the top level gains control and falls out of the handler, thus aborting the command interpreter and any nested procedures it has called. At this point, topLevel continues its loop and starts a fresh command interpreter.

The command interpreter itself has no knowledge that it is running under an interrupt handler.

If the command interpreter (or a procedure that it calls) enables interrupts, it has the first chance to handle them. If it chooses not to handle them, the procedure topLevel is notified of an interrupt.

Although the use of this handler appears to allow true asynchronous interrupt catching, the Scheduler gains control only during scheduled I/O (e.g., to the $tty if it is scheduled). Thus, this approach does not allow the program to abort another coroutine that is in a tight loop that does not do stream I/O or otherwise give up control to the STREAMS Scheduler. In such cases, calls to the $reschedule procedure could be inserted within the offending procedure to allow the Scheduler to check for interrupts.

8.5. Causing Keyboard Interrupts to Occur in a Child Process

When a stream is connected to a process through a PTY, the process can be signalled to interrupt using the procedure $writeStreamInterrupt.

In addition, PTY streams may have an associated character pty.$interrupt. If this character is not -1, it causes an interrupt if it is written to the PTY using $cWriteStream or as part of a $writeStream. It it has the value -1, it is not possible to interrupt the PTY process.

On some systems, a program running as a child process through a PTY may not be able to enable interrupt catching even though the program can enable them when running on an interactive terminal. On such systems, if the parent calls $writeStreamInterrupt, the child process is killed.

9. Low-level Stream I/O Procedures

Normally, programmers use remote procedure calls to implement distributed MAINSAIL programs. Occasionally, the RPC mechanism is not general enough, or the programmer may require more control over the stream than supported by RPC. The low-level stream I/O procedures are supplied to provide an escape mechanism for such cases.

```
NOTE:  Since STREAMS has not yet achieved a final form,
the low-level streams procedures are the most likely to
change.  RPC is sufficiently high-level that it is
unlikely to experience significant change.  Accordingly,
low-level streams procedures should be used only when
necessary, and the programmer should isolate such use so
that changes in their formulation will be easily tracked
in the application.
```

9.1. Overview of the Low-Level Stream I/O Procedures

Stream I/O consists of a flow of eight-bit character units. I/O to streams may be done to or from a supplied buffer, string, or an individual character. The type of buffer determines how the characters in the stream are interpreted and therefore how they are packed into the buffer or removed from the buffer.

9.1.1. Timeouts

Associated with each I/O procedure is a long integer timeout value, the number of milliseconds to wait for the operation to complete. The timeout value may be a positive long integer to cause the procedure to timeout after that number of milliseconds, or one of the constants shown in Table 9.1.1-1.

If timeout is omitted or $block is given, the procedure blocks indefinitely until something can be input/output or an error occurs. If timeout is greater than zero, it specifies a minimum amount of time in milliseconds to wait before returning a timeout failure. If timeout is $poll, the procedure returns immediately with either a success value or one of the error values described below, as applicable.

```
$poll           # Return immediately
$block          # Block indefinitely (infinite timeout)
```

Table 9.1.1-1. Special Timeout Values

If the system does not support timeout, i.e., if $systemSupportsTimeout returns false, all timeout values are treated as if they were $block and the procedure does not return until the I/O is complete or an error occurs.

9.1.2. Success and Failure of I/O Operations

The I/O procedures return a long integer that is nonnegative to indicate success and negative to indicate failure. Unless the bit errorOK is given to the procedure, however, errMsg is called with the fatal bit if a failure of any kind occurs.

The macros shown in Table 9.1.2-1 are defined to test the return values.

```
BOOLEAN
<macro>    $success (LONG INTEGER ee);
    # ee is return from stream proceedure; returns true
    # iff procedure succeeded

COMPILETIME
LONG INTEGER
<macro>    $error;          # General error occurred

COMPILETIME
LONG INTEGER
<macro>    $eos;            # End of stream occurred

COMPILETIME
LONG INTEGER
<macro>    $timedOut;       # Operation timed out
```

Table 9.1.2-1. Error Testing Macros

A return of $error indicates a general error. A return of $timedOut indicates that the I/O operation did not occur because the timeout value was exceeded. A return of $eos indicates that no more data are available on input, or that no more data can be accepted by the other end on output.

The character reading procedure $cReadStream returns integer instead of long integer values but is otherwise similar in this respect.

If an error return of any kind occurs, the field $lastInputError or $lastOutputError of the stream contains a description of the error. If the operation is neither input nor output, $lastInputError is set.

9.1.3. The General Error Return ($error)

The general error return occurs if the error does not fit one of the other error categories, or if the STREAMS package could not determine the precise reason for the failure. The error could be anything from a physical device error to an attempt to use an invalid stream handle.

9.1.4. The End-of-Stream Return ($eos)

The precise meaning of "end-of-stream" ($eos error) depends on the stream type, on whether the requested operations was input or output, and on the underlying implementation. The distinction between a normal $error and $eos might possibly be of some use to a program. When in doubt, treat $eos and $error the same way.

For TTY streams, in the input direction, $eos means that the user typed some system-dependent key combination to signal the end of an exchange (e.g., CTRL-D under BSD UNIX or CTRL-Z under VAX/VMS). Repeated $eos errors, however, might mean that no more input is available at all.

For socket streams, in both the input and output directions, $eos means that the process at the other end has "hung up" the connection for some reason.

For some implementations of some stream types, such as standard UNIX V.0 TTY streams, the STREAMS package may not be able to distinguish end-of-stream from a simple lack of data. On such platforms, the $eos return does not occur.

9.1.5. Timeout Return ($timedOut)

A return of $timedOut indicates that the I/O operation did not occur because the timeout value was exceeded. This error does not occur if a timeout value of $block (default) was supplied.

9.2. Octets

The term "octets" refers to eight-bit character units or their contents. The term differs from "characters" or "character units" in that octets are uninterpreted and untranslated; i.e., the stream operations on "characters" interpret their arguments or return values as text, whereas "octet" operations work on "raw bits".

All physical channels on which stream communication is based transmit information in the form of eight-bit units, so the octet is the quantum of information used by most stream I/O procedures.

9.3. Input: $readStream

```
LONG INTEGER
PROCEDURE    $readStream (POINTER($stream) st;
                          PRODUCES STRING s;
                          OPTIONAL BITS ctrlBits;
                          OPTIONAL LONG INTEGER timeout);

LONG INTEGER
PROCEDURE    $readStream (POINTER($stream) st;
                          CHARADR ca;
                          LONG INTEGER bufSize;
                          OPTIONAL BITS ctrlBits;
                          OPTIONAL LONG INTEGER timeout);
```

Table 9.3-1. $readStream (Generic) (continued)

```
LONG INTEGER
PROCEDURE    $readStream  (POINTER($stream) st;
                           ADDRESS a;
                           LONG INTEGER bufSize;
                           OPTIONAL BITS ctrlBits;
                           OPTIONAL LONG INTEGER timeout);
```

Table 9.3-1. $readStream (Generic) (end)

$readStream reads octets from the stream st into the destination string or buffer with maximum size given by bufSize.

The interpretation of the octets read from the stream (how they are packed into memory) depends on the the form of $readStream that is used. The string and charadr forms read text ($text mode), the charadr form with the $octet bit set reads octets, and the address form reads octets into complete storage units ($image mode).

It is not necessary to specify the bits $text or $image in ctrlBits because they are implied by the generic instance. However, when reading octets it is necessary to include the $octet bit, since octets are addressed by charadrs.

The precise interpretation of the octets in the stream is as follows:

- The text (string and charadr without the $octet bit set) forms interpret the incoming octets as characters and pack them into memory as strings are normally packed. By default, each read returns the next group of characters that is received from the stream, including eol characters but not necessarily an integral number of lines. bufSize for the charadr form is given in characters.

- The octet form packs the incoming octets into memory as full octets are normally packed. By default, the next group of octets that flow through the stream is read. bufSize is given in octets.

- The storage unit form packs the incoming octets into memory as full storage units. bufSize is given in storage units. By default, the next group of octets corresponding to an integral number of storage units are read.

At most one of the following buffering bits may be set:

- $unbuffered (default): $readStream waits until some octets are available. It returns all the available octets that fit into the supplied buffer, retaining any that do not fit for subsequent reads. In $text mode, the amount of text returned may not be an integral number of lines. End-of-line characters are included in the buffer where they occured in the input. In $image mode, $readStream always reads octets corresponding to an integral number of storage units.

- $line (for $text only): $readStream waits for a full line of text to become available. It then returns the characters in the line, including the end-of-line or line break character.

- $fillBuffer: $readStream waits until the supplied buffer has been filled completely. For the string form, $charsPerPage characters are read.

- $packet: $readStream reads a well-defined packet produced by the other end of the stream. The packet is guaranteed to be read and returned as a unit, rather than possibly arriving piecemeal. This mode is useful for communications between a client and server process, for example. Both the reader and writer must set $packet for the operation to work.

The buffering modes and defaults for each type of object read are summarized in Table 9.3-2.

Buffering Mode	$text	$octet	$image
$unbuffered	D	D	D
$line	X	-	-
$fillBuffer	X	X	X
$packet	X	X	X

Table 9.3-2. Buffering Modes (X = valid, - = invalid, D = Default)

If the read is successful, the number of characters or octets actually read is returned. Otherwise, a negative error value is returned.

On streams that buffer output opened for both input and output, the output is flushed with $flushStream (see below) before a read on the stream. This ensures that prompts or requests to

a user or another process are sent out to the other side before the read blocks waiting for a response.

The MAINSAIL procedure ttyRead acts similarly to the following stream calls:

```
$readStream($tty,tempString,$line);
read(tempString,resultString);
```

The second call to read removes the eol that $readStream returns as part of the line.

9.4. Output: $writeStream

```
LONG INTEGER
PROCEDURE    $writeStream
                        (POINTER($stream) st;
                         STRING s;
                         OPTIONAL BITS ctrlBits;
                         OPTIONAL LONG INTEGER timeout);

LONG INTEGER
PROCEDURE    $writeStream
                        (POINTER($stream) st;
                         CHARADR ca;
                         LONG INTEGER bufSize;
                         OPTIONAL BITS ctrlBits;
                         OPTIONAL LONG INTEGER timeout);

LONG INTEGER
PROCEDURE    $writeStream
                        (POINTER($stream) st;
                         ADDRESS a;
                         LONG INTEGER bufSize;
                         OPTIONAL BITS ctrlBits;
                         OPTIONAL LONG INTEGER timeout);
```

Table 9.4-1. $writeStream (Generic)

$writeStream writes the data contained in the specified buffer to the stream st.

The interpretation of the buffer (how it is converted into octets) depends on the the form of $writeStream that is used. The string and charadr (without $octet set) forms write text by encoding each character into an octet, the charadr form with $octet set in ctrlBits writes octets directly, and the address form writes storage units by converting them to octets in the standard way for the host system. The bufSize parameter is in character units for the charadr form and storage units for the address form.

It is not necessary to specify the bits $text or $image in ctrlBits because they are implied by the generic instance. However, when reading octets it is necessary to include the $octet bit because octets are addressed by charadrs.

At most one of the following buffering bits may be set:

- $unbuffered (default): $writeStream writes the octets and make them available to the other end of the stream immediately.

- $line ($text mode only): $writeStream writes the text in units of lines, including any eol characters, but (possibly) buffers up characters until an eol has been written before sending characters to the reader at the other end of the stream. The buffering is done if it is required for efficiency. Also, eol-like characters are converted to their proper form for the output device. For example, bare carriage returns and/or bare linefeeds are typically translated into carriage return-linefeed sequences if the output goes to a TTY and the device requires such a translation.

- $packet: $writeStream writes a well-defined packet to be read as a packet by the other end of the stream. The packet is guaranteed to be preserved as a unit upon arrival, rather than possibly arriving piecemeal. This mode must be used if and only if the reader calls the corresponding $readStream with $packet set. Clearly, this mode must be used as part of an overall communications protocol, e.g., between a server and a client or between two cooperating coroutines.

$writeStream returns a non-negative long integer on a successful write. Otherwise, a negative value is returned.

The system procedure ttyWrite produces results similar to:

```
$writeStream($tty,argStr,$line)
```

9.5. Single-Character I/O: $cReadStream and $cWriteStream

```
INTEGER
PROCEDURE     $cReadStream
                           (POINTER($stream) st;
                            OPTIONAL BITS ctrlBits;
                            OPTIONAL LONG INTEGER timeout);

LONG INTEGER
PROCEDURE     $cWriteStream
                           (POINTER($stream) st;
                            INTEGER c;
                            OPTIONAL BITS ctrlBits;
                            OPTIONAL LONG INTEGER timeout);
```

Table 9.5-1. $cReadStream and $cWriteStream

The procedures $cReadStream and $cWriteStream are similar to $readStream and $writeStream except that they read and write single characters or octets at a time. It is necessary to include the $text or $octet bit to indicate whether characters or octets should be read or written. It is illegal to specify $image, since MAINSAIL provides no portable way to address individual storage units.

The following bits are valid in ctrlBits:

- $text: On input, the next octet from the stream is interpreted as a character and returned in the rightmost $bitsPerChar bits of the result. If $line is set, the first character of a line may (possibly) not be returned until the entire line containing it has been received. On output, interpret the rightmost $bitsPerChar bits of the argument as a character and send it out as an octet. If $line is set, the character may (possibly) not be sent to the other end of the stream until an eol is output.

- $octet: On input, the next octet read from the stream is returned unchanged. On output, the rightmost eight bits of the argument are interpreted as an octet and sent out.

- The only legal buffering modes for $cReadStream and $cWriteStream are $unbuffered (the default) and $line (which affects only $text I/O).

On success, $cReadStream returns the actual octet or character value. On failure it returns one of the standard error codes $error, $eos, or $timedOut, except that the value is converted to an integer instead of a long integer (i.e., test for "cvi($error)", "cvi($eos)", or "cvi($timedOut)"). $cReadStream is the only stream I/O procedure that returns an integer instead of a long integer.

On success, $cWriteStream returns an arbitrary nonnegative value; on failure, it returns a negative value.

The buffer procedures $readStream and $writeStream are more efficient for reading many characters or octets. However, the character versions of these procedures are more convenient when program logic requires reading a single character or octet at a time.

9.6. Miscellaneous Operations: $flushStream and $clearStream

```
BOOLEAN
PROCEDURE    $flushStream
                         (POINTER($stream) st;
                          OPTIONAL BITS ctrlBits);

BOOLEAN
PROCEDURE    $clearStream
                         (POINTER($stream) st;
                          BITS ctrlBits);
```

Table 9.6-1. $flushStream and $clearStream

$flushStream forces all output done to the stream out so that it is available to the process or device at the other end. This action is of use if the implementation of the stream uses an internal buffer for efficiency. A read on the stream automatically causes the previously buffered output to be sent, but a program might want to ensure that all output has been sent prior to issuing an error message, for example.

$clearStream discards pending input (if input is set in ctrlBits) and/or output (if output is set in ctrlBits) on st. For example, if an editor reads an invalid command character from the terminal and prints an error message, it would be reasonable to call "$clearStream($tty,input)" so that all unread characters typed ahead by the user are discarded (since the user is likely to assume that the errant command was performed).

Both $flushStream and $clearStream return true if the operation was successful and false if the operation failed and errorOK was specified in ctrlBits. If errorOK was not set, they issue a fatal error.

9.7. Constants and Macros for Stream I/O

The following constants are provided to aid in writing portable MAINSAIL programs that manipulate streams:

- $optionalFirstEol: A character that sometimes precedes "last(eol)" in text read from streams, files transferred from other systems, or output from certain programs. The portable way to remove ends-of-lines from such text on all systems is to look for "last(eol)" and then discard the previous character if it is $optionalFirstEol.
- $error: The general error return value.
- $timedOut: A timeout error.
- $eos: An end-of-stream condition.
- $poll: A special timeout value that indicates immediate return if the operation would otherwise block.
- $block: A special timeout value that indicates an indefinite wait for an operation to complete. This is the default timeout for most procedures.

10. TTY Streams: TTYSTR

TTY streams communicate with TTY-like devices, including the controlling terminal, serial lines, and a parent process that is communicating with the current (child) process through a pseudo-terminal. TTY streams have special characteristics that often must be taken into account by the programmer, including keyboard interrupts, echo, and line editing.

Every implementation of the STREAMS package guarantees at least one TTY stream connected to the controlling terminal. This stream is always available in the STREAMS system variable $tty. In some cases, $tty may be a half-duplex TTY, that is, "$isHalfDuplex(tty)" is true. Half-duplex TTY's have restricted functionality (see Section 10.8).

Some systems support the ability to open an auxiliary serial port with terminal-like characteristics. On such systems, TTY streams to serial lines other than the controlling terminal may be opened using a name of the form "ttystr>xxx", where "xxx" is the (operating-system- and installation-dependent) name of the terminal line.

10.1. $tty

The controlling terminal is opened automatically by the STREAMS package and is available through the STREAMS system variable $tty. Input from the initial $tty stream comes from the keyboard and output usually goes to the screen, unless the primary input and output of the MAINSAIL process were somehow redirected by some external means.

File I/O done on files opened with the NTTY device module always goes through the current $tty stream. When cmdFile and logFile are opened through NTTY, they also go through the $tty stream.

If programs using $tty open or access the controlling terminal through non-STREAMS means, the mode setting logic for the controlling terminal may be confused and the redirection of $tty subverted. Such access therefore has undefined effects.

On systems that support the stream modules PTYPRO and SOCPRO, these may be used to start external processes in which $tty is not the physical terminal. The processes that may be run may possibly be restricted to MAINSAIL and specially written non-MAINSAIL processes on some systems.

10.2. Meaning of $eos on TTY Streams

On input, the $eos error return means that there is no more input available. This condition might be due to the user typing a special control character (such as CTRL-D under UNIX or CTRL-Z under VAX/VMS). In that case, a subsequent input may get more data. Alternatively, the $eos return might indicate a permanent condition, such as the end of a batch job input file.

On output, $eos means that no more data can be written.

10.3. TTY-Specific I/O Bits

TTY streams have some additional characteristics not present in other stream types. They are also unable to send or receive certain octet values (i.e., special control characters) unless some special bits are specified to the stream I/O procedures. The characteristics of TTY streams include:

- echo and line editing: Certain default actions take place when reading a line from a TTY, including the echo of typed characters back to the output side of the stream (i.e., to the "screen"). Also, certain characters typed by the user are interpreted as line editing functions for correcting mistakes (e.g. delete or backspace). However, at times these defaults are not desired.

- interrupt characters: Certain character codes are interpreted in a special way by the operating system. When these characters are received by a TTY, they cause "interrupts" to occur. For example, CTRL-C on the controlling terminal might abort the program doing the read. However, at times these interrupt characters must be turned off so that all characters may be read by the program.

- flow control: TTY streams often use distinguished character codes (e.g., XON/XOFF) to signal the writer that the reader is ready to accept more data. Flow control prevents the receiver's buffer from overflowing and is commonly used by interactive terminals, modems, and operating systems.

- transmission speed: Various TTY devices are capable of transmitting at different speeds (baud rates). The sender and receiver must agree on the rate.

When $text is read from a TTY stream in $line mode, and no special TTY-specific bits are set, reading occurs in units of lines, line editing characters are available to the user at the keyboard to correct mistakes on input, and the characters read are echoed to the output end of the stream.

When any of the buffering bits other than $line is used for TTY input, both line editing and the echoing of input to the output are disabled until a read is done from the TTY stream in which $line is specified.

When $text is written to a TTY stream in $line mode, eol characters are modified as necessary to cause the cursor to move to the beginning of the next line. Typically it is necessary to supply both a carriage return character and a linefeed character to the actual physical terminal. When output is done in a buffering mode other than $line, e.g., $unbuffered, no such manipulation of eol characters takes place. It is then the program's responsibility to ensure that the desired sequence of characters is sent. That sequence is available as the $stream field $unbufferedEol.

For example, under UNIX, the string eol contains the single character linefeed. If this alone were output to most terminals, the cursor would not move back to the first column of the screen; i.e.:

```
$writeStream($tty,"Hi" & eol & "Hi")
```

would produce:

```
Hi
  Hi
```

while either:

```
$writeStream($tty,"Hi" & eol & "Hi",$line)
```

or:

```
$writeStream($tty,"Hi" & $tty.$unbufferedEol & "Hi")
```

would produce:

```
Hi
Hi
```

Aside from the side effects of echoing and eol manipulation associated with $line, the $line and $unbuffered bits work for TTY's as they do for all other streams. The $fillBuffer and $packet buffering modes are of little use for TTY streams because such streams are unreliable when connected to a physical terminal line. However, these modes might be useful over a TTY stream that is connected to a parent process instead of a physical terminal.

System-specific interrupt characters and flow control characters introduce a slight complication into programming a TTY stream. By default, interrupt characters have a special effect and are

not passed to the program. The flow control characters are not available to the program, either for reading or writing.

Unless some special bits are set for TTY stream I/O, the programmer must be careful not to expect to read octets that correspond to interrupt or flow control characters or to write flow control characters and have them received at the other end of a TTY stream. Since these characters are usually non-printing characters, normal applications do not need to worry about this limitation. However, special applications such as screen editors and terminal emulators must take special action.

To vary these characteristics from the default, additional ctrlBits bits are provided for the stream input functions. The additional bits are:

- $noInterrupt: Read characters that normally cause keyboard interrupts (e.g., CTRL-C) and return them as normal characters without causing the usual interrupt action. This mode is useful for implementing terminal emulators. It is set automatically for $image reads. It is illegal in combination with $line.

- $noFlow: Turn off flow control and read the flow control characters as normal characters. This mode is useful for implementing multiwindow terminal emulators that wish to read XON/XOFF and apply them selectively to each window. It is set automatically for $image reads.

- $flow: Turn on flow control. If neither $flow or $noFlow is set, the stream module sets flow control in a device-dependent manner. In particular, for the controlling terminal, if neither bit is given, the flow control is set the way the user had it set up prior to running MAINSAIL.

These bits are ignored for non-TTY streams or in implementations where they cannot be supported.

For example, to read a character without echo and without waiting for an eol, "$cReadStream($tty,$text!$unbuffered)" could be used, provided that $tty is not a half-duplex terminal. The program may want to echo the character it read. If it does not do so explicitly, the user does not see any echo.

The system procedure "ttycWrite(char)" functions similarly to "$cWriteStream($tty,char,$text!$line)". To cause char to be displayed immediately rather than waiting for an eol (full-duplex systems only), "$cWriteStream($tty,char,$text!$unbuffered)" could be used instead. However, when $unbuffered is set, it is the responsibility of the program to make sure that the sequence of characters $tty.unbufferedEol is output at the end of a line. This character sequence is not necessarily the same as the string constant eol.

10.4. TTY-Specific Procedures

```
LONG INTEGER
PROCEDURE    $getBaudRate
                        (POINTER($stream) st;
                         OPTIONAL BITS ctrlBits);

BOOLEAN
PROCEDURE    $setBaudRate
                        (POINTER($stream) st;
                         LONG INTEGER baud;
                         OPTIONAL BITS ctrlBits);

LONG INTEGER
PROCEDURE    $writeStreamBreak
                        (POINTER($stream) st;
                         OPTIONAL BITS ctrlBits;
                         OPTIONAL LONG INTEGER timeout);
```

Table 10.4-1. TTY-Specific Procedures

The procedure $getBaudRate returns the baud rate of the TTY stream st, 0L if the rate is unknown, and negative failure values if st is not a valid stream or some other error occurs.

The procedure $setBaudRate sets the baud rate of the TTY stream st to baud, returning true if it was successful and false otherwise.

Setting errorOK in ctrlBits for these functions suppresses a fatal error message; the error text is then put in st.$lastInputError.

The procedure $writeStreamBreak sends a break on a TTY stream, returning a nonnegative value if the function is implemented and succeeds, or a negative failure value otherwise. Breaks are typically read as one or more null characters (i.e., $nulChar). Setting errorOK in ctrlBits suppresses a fatal error message; the error text is then put in st.$lastOutputError.

10.5. TTY-Specific Fields of $stream

The TTY-specific fields of $stream give the character sequences corresponding to specially interpreted characters (normally these are control characters) that may be typed by the user to cause special actions.

```
STRING
    $unbufferedEol; # How to write eol in $unbuffered
                    # mode. This is typically a two-
                    # character string, even though
                    # length(eol) = 1.

INTEGER
    # Characters received in $unbuffered mode when:
    $returnKey,     # A return key is pressed
    $linefeedKey,   # A linefeed key is pressed
    $eofIndicator,  # An end-of-file character is typed

    # Line editing characters for this TTY:
    $erase,         # Erase the most recent character sent
    $altErase,      # Erase the most recent character sent
    $wordErase,     # Erase the most recent word sent
    $lineErase,     # Erase the entire line
    $reprint,       # Reprint the current line being input
    $quoteNext,     # quote the next character

    # Flow control characters for this TTY:
    $startOutput,   # An XON (e.g., CTRL-Q)
    $stopOutput;    # An XOFF (e.g., CTRL-S)

    # Interrupt characters for this TTY:
    $interrupt,     # The interrupt character
    $discardOutput; # Clear (discard) pending output
```

Table 10.5-1. TTY-specific Fields of $stream

If any of the functions implied by these fields is not supported by a given system, the corresponding characters are -1. The only fields that are guaranteed to be defined are $unbufferedEol, $returnKey, and $erase.

Using these character field definitions, a program that wishes to read characters in a mode other than $line may intelligently interpret the special incoming character sequences according to the host system conventions to achieve each of these functions.

The $startOutput and $stopOutput characters tell a program what characters are reserved for flow control (unless $noFlow mode is in effect).

The complete set of characters reserved for interrupts may not be determined. The character $interrupt is the standard interrupt character (the one that may possibly be trapped by the STREAMS package and that otherwise causes program termination) if such a character is available for this system and TTY. The character $discardOutput is the character the user may type to discard the pending output to the terminal. Neither of these characters may be read from the TTY unless $noInterrupt is set.

10.6. Is My $tty Interactive?

A program may determine whether its $tty has the characteristics of a true terminal or not. If not, it should not expect interaction with a real user.

```
IF $isaTty($tty) THEN # I am interactive or PTYPRO
EL                    # I am batch or SOCPRO
```

On UNIX, a program for which "$isatty($tty)" is false might also be running with its standard input coming from a pipe or file set up by the shell.

This test works if the system supports a means to determine the difference between an interactive and "batch" terminal. Otherwise, "$isatty(st)" always returns true.

10.7. Am I a Cooperative Child?

A program may determine if its parent expects to talk to it through the $parent stream by checking if $parent is nullPointer, e.g.:

```
IF $parent THEN # I am a cooperative child
```

Cooperative children may or may not have an interactive TTY.

10.8. Half-Duplex TTY Streams

On half-duplex TTY streams, the only allowed buffering bit for input is $line. For output, both the $unbuffered (default) and $line bits are supported, but $unbuffered text output may not behave differently from $line output. The programmer must use adaptive programming tests in a portable program to find alternative methods if the program would normally perform $unbuffered, $packet, or $fillBuffer I/O and it is to be run on a half-duplex system, e.g.:

```
IF NOT $isHalfDuplex(st) THEN
    # Full-Duplex Case
    $readStream(st,...,$unbuffered);
EB # Half-Duplex Case: Cannot turn off echo
    positionCursorToTopOfScreen(st); # user-supplied function
    $readStream(st,...,$line);
    repositionCursor(st) END;        # user-supplied function
```

10.9. PTY Streams

When PTYPRO is used to start a child coroutine (see Chapter 6), the child's $tty is set up to communicate directly with the parent's PTY stream. As far as the child is concerned, its $tty has all of the characteristics of an interactive terminal, including echo, interrupts, and flow control. However, in fact, what it reads is supplied by the parent coroutine that opens the PTY, and what it writes is supplied to the parent on the PTY.

PTY is an acronym for "pseudo-TTY". The interaction of the parent program with the PTY is much like the interaction of a user with a terminal keyboard and screen. It is the "other end" of the child process's $tty.

Parent Coroutine		Child Coroutine
$writeStream(pty,...);	===>	$readStream($tty,...);
$readStream(pty,...);	<===	$writeStream($tty,...);

Table 10.9-1. Interaction Between PTY and $tty

The coroutine that opens the PTY stream is responsible for simulating a person sitting at a terminal. It must write characters to the PTY to simulate typing at the keyboard and read

characters from the PTY that would normally appear on the screen (including echo of the characters it wrote!).

On systems that support process creation and control, PTY streams may be used to create and talk to an independent program running in a separate (child) process external to the creating (parent) process. On systems that support transparent pseudo-terminals, any program may be run through the PTY without restriction, not just MAINSAIL programs. On such systems, the standard system "shell" (command interpreter) may be run.

To the child process that is running through the PTY, everything appears as if it were connected to a true interactive terminal. The child process may turn echo on and off, read characters in $line or $unbuffered mode, interpret interrupt characters specially or not, and interpret flow control specially or not as it chooses. The parent process itself does not use these modes when reading from or writing to the PTY.

PTY streams are used to communicate with child processes when the details of the interaction are unknown to the program that opens the PTY. Typically, the program that opens a PTY serves merely as an intermediary that accepts keystrokes from a user (passing them on to the child process) and displays the resultant input from the PTY on the screen. For example, a window manager can use PTY's to implement a window that talks to a separate job.

The only buffering mode that makes sense in general for both input and output on PTY's is $unbuffered. The output is made available to the child process immediately. The child process may or may not choose to buffer when it does its input, but that is entirely separate. PTY input should be done in $unbuffered mode since it is based on the available characters written by the child. Because the exact interaction expected by the child is generally unknown, it is not generally useful to perform input from the PTY in $line mode (doing so could cause the parent to fail to display, e.g., a prompt not terminated by eol, until after the text responding to the prompt had been set, an undesirable interaction).

The use of PTY streams to control child processes is described in Section 6.4.

10.9.1. Meaning of $eos on PTY Streams

On both input and output, $eos means that the process controlled by the PTY is no longer alive.

There are no PTY-specific I/O bits. The only buffering mode supported is $unbuffered.

10.9.2. PTY-Specific Procedures

```
LONG INTEGER
PROCEDURE   $writeStreamBreak
                        (POINTER($stream) pty;
                         OPTIONAL BITS ctrlBits;
                         OPTIONAL LONG INTEGER timeout);

LONG INTEGER
PROCEDURE   $writeStreamInterrupt
                        (POINTER($stream) pty;
                         OPTIONAL BITS ctrlBits;
                         OPTIONAL LONG INTEGER timeout);
```

Table 10.9.2-1. PTY-Specific Procedures

The procedure $writeStreamBreak sends a break on a PTY stream, returning a nonnegative value if the function is implemented, or a negative failure value otherwise. Breaks are typically read on the other end as one or more null characters (i.e., $nulChar). Setting errorOK in ctrlBits suppresses a fatal error message; the error text is then put in pty.$lastOutputError.

The procedure $writeStreamInterrupt signals an interrupt to the child process. If the child has not enabled itself to catch the interrupt or if interrupt catching is not implemented, the child will probably be killed. The bit errorOK suppresses error messages. An optional timeout may be given to return early if the procedure blocks.

In addition, a PTY stream pty has an associated character pty.$interrupt. If this character is nonnegative, it causes an interrupt if it is written to the PTY using $cWriteStream or as part of a $writeStream, e.g., "$cWriteStream(pty,pty.$interrupt)".

10.9.3. PTY-Specific Fields of $stream

The PTY-specific fields give the character sequences corresponding to specially interpreted characters (normally these are control characters) that may be typed by the user to cause special actions. These are the same as the TTY-specific fields (see Table 10.5-1).

If any of the fields is not supported by a given system, the corresponding character is -1. The only fields that are guaranteed to be defined are $unbufferedEol, $returnKey, and $erase.

11. MEMORY Streams: MEMSTR

Memory streams are used for two-way communication between coroutines. They are created in pairs with the stream module MEMSTR. Both coroutines typically know that they are communicating with each other so that an understood communications protocol can be used.

Output to $openStream's stream s1 is read as input on s2, and output to s2 is read as input on s1. The Scheduler (see Chapter 7) ensures that the coroutines that read and write the streams are scheduled appropriately.

Memory streams are useful for serialized communication of text, octets, or storage units between two or more coroutines running in the same MAINSAIL process. They are a low-overhead simulation of the use of socket streams for interprocess communication. Closely coupled coroutines that share arbitrarily complex data structures should probably use semaphores instead of memory streams.

Memory streams do not have semantics normally associated with TTY streams: no echo occurs, there are no special keyboard interrupt characters, and there is no explicit flow control or baud rate.

There are no special bits, fields, or procedures associated with memory streams. The $eos return means that the other end of the stream has been closed.

It is invalid to give a host name for a MEMSTR stream.

```
NOTE: MEMORY streams are a special case of RENDEZVOUS
      streams for use when both tasks are running in the
      same process.
```

Appendix A. Server Installation Instructions

The XIDAK service named "gensrv" is used to support several STREAMS features, including:

- Remote file access using the NET device module
- Remote stream gateway using NETSTR
- Server status using the SRVINF module

For these features to be available, the "gensrv" server process must be started on each node and a service protocol table must be created containing the names of the host systems and the servers that are running on each host.

To run the "gensrv" server, it is wise to use the special MAINSAIL bootstrap named "gensrv". This bootstrap is built during the MAINSAIL installation process. It is configured to use less memory than normal interactive MAINSAIL processes.

A.1. The Format of the XIDAK Service Protocol Table

The server/client implementation currently depends on the availability of a file describing the services available on each host. The logical file name "(service protocol table)" must be defined, typically with a MAINEX "ENTER" subcommand in the "site.cmd" command file for each MAINSAIL installation.

The need to maintain service protocol tables may eventually be replaced by a XIDAK name server. When this happens, a program interface will be supplied to update the service protocol table throughout a local network.

A typical service protocol table is shown in Example A.1-1.

The commands in the service protocol table are described below.

A.1.1. "MYHOST hostName"

The "MYHOST" entry, not shown in Example A.1-1, specifies the name of the host node under which MAINSAIL is running. This entry is needed only on systems that do not provide a

```
HOSTNAME Apollo apollo s
HOSTNAME Poseidon poseidon p

HOSTPROTOCOL Apollo tcp
HOSTPROTOCOL Poseidon tcp

SERVICE gensrv Apollo/tcp Poseidon/tcp
SERVICE srv Poseidon/tcp
SERVICE ucrsrv Apollo/tcp

DEFAULTPROTOCOL tcp
```

Example A.1-1. Typical Service Protocol Table

system call that returns the current host's name. It should be omitted when possible, since leaving it out allows the same table to be shared among several hosts.

A.1.2. "HOSTNAME officialName alias1 alias2 ... aliasn"

The "HOSTNAME" entry declares a host that is directly accessible from the current system by some network protocol. There must be one "HOSTNAME" entry per accessible host. The first name is the "official" or canonical name by which that host is known, and the remaining names are acceptable aliases for the official name.

A.1.3. "HOSTPROTOCOL hostName protocol1 protocol2 ... protocoln"

Each "HOSTNAME" entry must have a corresponding "HOSTPROTOCOL" entry to describe the network protocols through which that host may be accessed. The protocols are valid STREAMS protocol modules by which the access is made. Generally these names match the network methodology (e.g., TCP for TCP/IP, VMSMBX for VAX/VMS named mailboxes, etc.). The list should be ordered in preference, with the most preferred method listed first.

Available network protocol modules are listed in Appendix B.

A.1.4. "SERVICE serviceName hostName/protocol ..."

For each service that is to be accessed or provided by a MAINSAIL program, there must be a "SERVICE" entry. This entry specifies that the named service (e.g., "gensrv", "ucrsrv") is available on the named hosts under the specified network protocols (the hosts and protocols are associated in pairs).

A.1.5. "DEFAULTPROTOCOL protocol1 protocol2 ... protocoln"

The "DEFAULTPROTOCOL" entry lists the default network protocols to try if the host is not explicitly declared. This entry is optional.

A.2. Host-Dependent Service Tables

On some systems, such as UNIX, it is necessary to enter service names in a host-dependent table or file in addition to entering them in the XIDAK table. For example, under BSD UNIX, the file "/etc/services" must be updated to contain the service name and port number for each service to be supplied or accessed. On SunOS, the Yellow Pages server must be used instead of directly editing the file "/etc/services". Consult the documentation provided by the operating system for details.

A.3. Installing the "gensrv" Server and the Service Protocol Table

The steps required for installing "gensrv" on a host are:

1. Create a service protocol table (it may be possible to share the same table on several different hosts).

2. Add an "ENTER" subcommand to the "site.cmd" startup command file (for each installation of MAINSAIL being used) that points to the service protocol table (see below).

3. Declare the "gensrv" service in an operating system file, if required (e.g., in "/etc/services" under BSD UNIX).

4. Start the "gensrv" process (using the "gensrv" bootstrap) as a background process. It may be desirable to set up the operating system to do this whenever it is rebooted, e.g., in an operating-system-specific command file.

These steps must be performed on all systems on which clients or servers are to be run. The steps are described in detail below.

A.3.1. Create a XIDAK Service Protocol Table

The minimal service protocol table required to have general access between hosts A and B is shown in Example A.3.1-1. This example assumes that the TCP protocol is being used. Other protocols may be appropriate, and more than one protocol may be supplied for a host or service on a host.

```
HOSTNAME A
HOSTNAME B

HOSTPROTOCOL A tcp
HOSTPROTOCOL B tcp

SERVICE gensrv A/tcp B/tcp
```

Example A.3.1-1. Minimal General Service Protocol Table between Hosts A and B

A.3.2. Add "ENTER" Subcommands to the MAINSAIL Site-Specific Startup Command File ("site.cmd")

The service protocol table is accessed by MAINSAIL through the logical file name "(service protocol table)". An "ENTER" subcommand for this logical name must be made, e.g., in the MAINSAIL startup command file "site.cmd", on all MAINSAIL installations that are to be servers or clients, as shown in Example A.3.2-1.

```
ENTER (service protocol table) /etc/mslhosts
```

Example A.3.2-1. "ENTER" in "site.cmd" for Using Servers

A.3.3. Modify the Operating System Services Table

Some operating systems require that the service "gensrv" be declared in a system table. For example, under BSD UNIX, the file "/etc/services" must contain an entry similar to the one shown in Example A.3.3-1. This line tells the host system that the "gensrv" server is available under port number 2000 using TCP (note that this kind of port, a TCP port, is not the same thing as a MAINSAIL port as represented by a $port record). You may have to choose a different number if 2000 is already used as a local service, but be sure to choose the same number on each system of the network. The number may be chosen arbitrarily within the constraints on TCP port numbers; check your operating system documentation for details.

```
gensrv          2000/tcp          # XIDAK GENSRV Server
```

Example A.3.3-1. Declaring the Service "gensrv" to BSD UNIX

A.3.4. Arrange for "gensrv" to be Started as a Background Process

The MAINSAIL bootstrap "gensrv" starts the XIDAK GENSRV server. Because this bootstrap is configured to use less memory than an interactive MAINSAIL, it is better to use it than the standard MAINSAIL bootstrap.

The method of starting a background process varies from system to system. "gensrv" should generally run as a "superuser" or "root" job so that it can access all files. It should also run connected to a directory with no important files in it, just for saftety.

Under BSD UNIX, for example, the file "/etc/rc.local" may be edited to start GENSRV as a local daemon by adding lines as shown in Example A.3.4-1. This example assumes that MAINSAIL resides in the directory "/usr/mainsail". The entry in "/etc/rc.local" causes "gensrv" to be started automatically whenever the computer system is booted.

GENSRV may be started manually under UNIX as follows:

1. Become super user.

2. Connect to "/tmp" (e.g., do "cd /tmp").

3. Issue the command "nohup /usr/mainsail/gensrv &".

```
if [ -f /usr/mainsail/gensrv ]; then
(echo -n ' gensrv'; cd /tmp; /usr/mainsail/gensrv&) \
    >/dev/console
fi
```

Example A.3.4-1. Starting GENSRV Automatically Under BSD UNIX

"gensrv" must run on all nodes where the desired remote files are located.

Appendix B. Available Network Protocol Modules

Table B-1 lists the currently available stream protocol modules for interprocess communication.

Module	Systems	Supports	Can Detect End of Stream
DECNET	VAX/VMS	DECNET; server must have SYSNAM privilege set	YES
TCP	UNIX	TCP/IP	YES
TCPWOL	VAX/VMS	Wollongong's WIN/TCP	YES
VMSMBX	VAX/VMS	VAX/VMS named mailboxes	NO

Table B-1. Protocol Modules and Their Characteristics

Appendix C. System-Specific Support For Streams

Table C-1 lists the currently implemented support for various stream- and Scheduler-related features.

Plat. Abbr.	TTY Int.	Sched. TTY	Sched. Non-TTY	Sched. TO's	SOC.	PTY.	SERV.
aeg	*	*	*	*	*	*	*
aix	+	?	+	+	+	?	+
alnt	+	+	+	+	+	+	+
cms	+	+	+	+	-	-	+
dgux	*	*	*	*	*	*	*
emb	?	?	?	?	?	?	?
hp20	*	*	*	*	*	*	*
hp38	+	+	+	+	+	+	+
hpux	*	*	*	*	*	*	*
ip32c	+	+	+	+	+	+	+
ipsc2	+	+	*	*	*	*	*
ix20	*	*	*	*	*	*	*
ixat	+	+	+	+	+	+	+
ixfpa	*	*	*	*	*	*	*
mvux	?	?	?	?	?	?	?
ros	*	*	*	*	*	*	*
spix	+	+	+	+	+	+	+
sun2	*	*	*	*	*	*	*
sun3	*	*	*	*	*	*	*
sun38	*	*	*	*	*	*	*
sun4	*	*	*	*	*	*	*
sw38	*	*	*	*	*	*	*
ultrx	*	*	*	*	*	*	*
uts5	?	?	?	?	?	?	?
xcms	+	+	+	+	-	-	+
vms	*	*	*	*	*	-	*

Table C-1. Features Supported by STREAMS Implementations (continued)

Key:

```
    * = IMPLEMENTED
    + = POSSIBLE (BUT NOT IMPLEMENTED)
    ? = UNKNOWN WHETHER POSSIBLE
    - = IMPOSSIBLE TO IMPLEMENT

    Plat. Abbr. =         Platform abbreviation
    TTY Int. =                    TTY interrupts
    Sched. TTY =          Scheduled TTY stream
    Sched. Non-TTY =      Scheduled streams other than TTY
    Sched TO's =          Timeouts when Scheduling
    SOC. =                SOCPRO child processes
    PTY. =                PTYPRO child processes
    SERV. =               SERVICE service streams
```

Comments:

(1) Standard UNIX SVR0 and SVR2 do not support efficient TTY scheduling.

(2) Standard UNIX SVR0 and SVR2 have no network communication facilities and no PTY jobs.

(3) The UNIX SVR3 concept of TTY streams is currently unimplemented by all known SVR3 UNIX ports. The STREAMS package could support scheduled TTY streams using a busy poll loop with a small timeout.

(4) UNIX SVR4 promises to support all stream functions but is not yet available for testing as of this writing.

Table C-1. Features Supported by STREAMS Implementations (end)

Appendix D. Extended C RPC Client Example

The examples in this section demonstrate a greater variety of argument types used in a C client than shown in Chapter 4. Example D-2 is a C client that calls a MAINSAIL server with an interface procedure xxx with the header shown in Example D-1. The MAINSAIL remote module is called R01.

```
PROCEDURE xxx (
        MODIFIES STRING ARRAY(1 TO 11) Ary1;
        PRODUCES LONG INTEGER ARRAY(-2 TO 2) Ary2;
        PRODUCES STRING ZS3;
        CHARADR PdfBuf4Pdf; LONG INTEGER PdfBuf4Length;
        CHARADR TxtBuf5; LONG INTEGER TxtBuf5Length;
        ADDRESS DatBuf6; LONG INTEGER DatBuf6Size;
                PRODUCES LONG INTEGER DatBuf6Length;
        USES STRING S7;
        USES LONG INTEGER Li8);
```

Example D-1. MAINSAIL RPC Server Interface Procedure Header

```
#include "mrpc.h"
#include "r01cli.h"

MRPCMOD *rem;
```

Example D-2. C Client That Calls MAINSAIL Remote Module (continued)

```
newAry1(a,typeCode,lb1,ub1,elements)
ARRAY *a;
int typeCode,lb1,ub1;
char *elements;
{
a->ary_dims = 1;
a->ary_type = typeCode;
a->ary_lb1 = lb1; a->ary_ub1 = ub1;
a->ary_first_elem = elements;
}



newAry2(a,typeCode,lb1,ub1,lb2,ub2,elements)
ARRAY *a;
int typeCode,lb1,ub1,lb2,ub2;
char *elements;
{
newAry1(a,typeCode,lb1,ub1,elements);
a->ary_dims = 2;
a->ary_lb2 = lb2; a->ary_ub2 = ub2;
}



newAry3(a,typeCode,lb1,ub1,lb2,ub2,lb3,ub3,elements)
ARRAY *a;
int typeCode,lb1,ub1,lb2,ub2,lb3,ub3;
char *elements;
{
newAry2(a,typeCode,lb1,ub1,lb2,ub2,elements);
a->ary_dims = 3;
a->ary_lb3 = lb3; a->ary_ub3 = ub3;
}
```

Example D-2. C Client That Calls MAINSAIL Remote Module (continued)

```
main (argc,argv)
int argc;
char *argv[];
{
int fd;
char host[100],serv[100],emsg[100];
char *hostS;
char *servS;

long xxxx;
ARRAY Ary1;
char **eAry1;
ARRAY Ary2;
long *eAry2;
char *ZS3;
char *PdfBuf4Pdf; long PdfBuf4Length;
char *TxtBuf5; long TxtBuf5Length;
char *DatBuf6; long DatBuf6Size;
long DatBuf6Length;
char *S7;
long Li8;

if (argc <= 2) {
    printf("Host: "); gets(host); hostS = host;
    printf("Server: "); gets(serv); servS = serv; }
else { hostS = argv[1]; servS = argv[2]; }

if ((fd = connserver(hostS,servS)) < 0) exit(1);

if (! (rem = r01_init(fd,0,0,emsg))) {
    printf("Could not init: %s\n",emsg);
    exit();
    }

eAry1 = (char **) calloc(11,sizeof(char *));
newAry1(&Ary1,stringCode,1,11,eAry1);

... set up parms...
```

Example D-2. C Client That Calls MAINSAIL Remote Module (continued)

```
r01_xxx(rem,&Ary1,&Ary2,&ZS3,PdfBuf4Pdf,PdfBuf4Length,
        TxtBuf5,TxtBuf5Length,DatBuf6,DatBuf6Size,
        &DatBuf6Length,S7,Li8);

... use results...

Ary1.ary_alloc_status = ARY_DATA_DYN | ARY_STRING_DYN;
dispose_array(&Ary1);
Ary2.ary_alloc_status = ARY_DATA_DYN;
dispose_array(&Ary2);
if (ZS3) { cfree(ZS3); }
}

r01_final(rem);
close(fd);
}
```

Example D-2. C Client That Calls MAINSAIL Remote Module (end)

Appendix E. Remote Streams: NETSTR

The NETSTR stream module is used internally to implement opening streams on other nodes and to support gateways.

The NETSTR stream module opens a stream on another computer system. It may be used in conjunction with any stream module, including the stream modules SOCPRO and PTYPRO that create processes. Thus, NETSTR can be used to create and control a process on another node in the network.

For NETSTR to be used, the service "gensrv" must be available on the desired host system and the gensrv process must have been started on that system (see Appendix A).

Remote streams are opened through the NETSTR stream module, with the syntax:

```
netstr(hostName)>remoteStreamName
```

```
NOTE: As of this writing, the only accepted syntax is:
      netstr>hostName:remoteStreamName
```

E.1. Opening a Remote TTY Device

To open a TTY stream named "/dev/ttyb" on a host "a":

```
$openStream(lpt,"netstr(a)>ttystr>/dev/ttyb",...);
```

E.2. Explicit Gateway Mechanism

Because the syntax of NETSTR allows nested calls to NETSTR, it provides an explicit gateway mechanism between nodes in networks linked by a common node.

For example, suppose that nodes A and B are part of network N1, and nodes B and C are part of network N2. Node B is a gateway node because it is part of both networks.

A process running on node A can create a process on node C using the stream name:

```
"netstr(B)>netstr(C)>socpro>...."
```

The process running on node A asks node B to create a process on node C.

E.3. Underlying Implementation

The NETSTR stream module establishes communication with the "gensrv" server on the remote host and requests that the stream be opened. Depending on the implementation and the type of stream being opened, NETSTR may perform all operations through "gensrv", or it may establish independent communication that does not go through the remote "gensrv".

Appendix F. A Guide to Remote File Access Using NET

F.1. Use of the NET Device Module

NET is a MAINSAIL device module for accessing files on other computers through a network. The general form of a file name that uses NET is:

```
net(hostName)>hostPath
```

e.g.:

```
net(george)>/usr/bob/foo
```

(">" is shown for the device module separator, $devModBrk; the actual character may differ on some operating systems).

The hostName field is the name by which the desired system is known to the machine on which MAINSAIL is running. The storageUnitSize is the size of storage units in the dataFile to be accessed. It is optional and applies only to data files. The hostPath is the file name in the syntax expected by the host system. The hostPath must be a complete path name, since there is no concept of a current directory on a remote system.

The normal MAINSAIL file I/O routines work as usual. The fact that the file resides on another host is hidden from the user, except for speed of access. Thus, for example, files on remote systems may be edited by the MAINSAIL editor or compiled by the MAINSAIL compiler, and compiler output may be directed to a remote system.

[NOTE: Due to a lack of generality in the $rename system procedure, the compiler currently is not able to output an intmod to a remote system. This problem may be fixed eventually.]

Internally, the NET module reads and writes the file one buffer at a time. Thus it is not necessary to wait for the entire file to be transmitted to read or write it. The remote file never exists in the local file system.

On systems that do not provide operating system file versions, the server that provides access to the system's files prevents more than one user from opening the same file for conflicting purposes through NET. Thus, if two processes attempt to open a UNIX file "net(foo)>/usr/bob/bar" for writing, the second process gets an error. However, no check for

simultaneous local access is made unless the operating system supplies this check as part of its standard file I/O.

F.2. Execution of Modules Stored Remotely

Modules that are stored in files or libraries on a remote system may be executed through NET simply by arranging for the associated file name to reference the remote file. The MAINEX "SEARCHPATH" and "ENTER" subcommands may be set up to access modules on another system through NET, e.g.:

```
SEARCHPATH *.olb *.olb net(central)>/usr/mainsail/*.olb
ENTER graflib net(george)>/usr/george/lib/grafix.olb
```

F.3. Syntactic Sugar for Remote File Access

To make entering of remote file names easier, define searchpaths for remote systems of the form:

```
SEARCHPATH george:* net(george)>*
```

This allows files on the node "george" node to be accessed with the prefix "george:".

F.4. Known Restrictions and Limitations

NET is available only for systems on which scheduled advanced STREAMS is supported.

Before NET can be used, servers must be installed and service protocol tables must be set up. The installation of the remote file system is described in Appendix A.

As currently implemented, there is no notion of security or file protection in NET. Because the server typically runs as root, any file on the remote system can be accessed through NET. Also, the default file protection for files created through NET reflects that of the server process, which may cause problems if the file is later accessed locally on that system by a non-root user.

Each host system imposes some limit on the number of simultaneous open files a process may have. Since all files accessed remotely at a given host are accessed through a single process on that host, the maximum number of remote files open on a given system is limited and may easily be exceeded. For example, the server running on SunOS uses three file handles for STDIN, STDOUT, and STDERR, MAINSAIL itself uses one for its system library, the port for accepting clients uses an additional one, and each remotely opened file uses two files (the

socket stream and the actual file being accessed). To verify files, $fileInfo is used which requires at least one additional file. Since at most 31 files may be opened by a process under SunOS Version 3, only 12 files may be open simultaneously through NET to a particular SunOS host.

If NET is used to open a non-disk file that blocks for long periods (e.g., a terminal), all clients accessing the same server are blocked.

The check for simultaneous access of files through NET currently uses the $fullPathName field returned by $fileInfo. On some systems, it is possible for the same physical file to have different $fullPathName values (e.g., hard links under UNIX). On such systems, the simultaneous access check done in the server may fail to detect the collision, with possible damage to the file being accessed twice.

Appendix G. XIDAK STREAMS Applications and Utilities

This appendix describes software written by XIDAK that uses STREAMS but is not necessarily part of the STREAMS package. Contact XIDAK for further information.

G.1. Technology Database Management System: TDB

The TDB database management system is a relational database system for distributed heterogeneous computing environments. For more information on TDB, contact XIDAK for the relevant documentation.

G.2. Network File Access: NET Device Module

The MAINSAIL device module NET allows a MAINSAIL program to access files on other systems as if the files were local. It works by talking to a server process ("gensrv") on the remote system. Refer to Appendix F for more information.

G.3. Bug Tracking System: UCRSYS

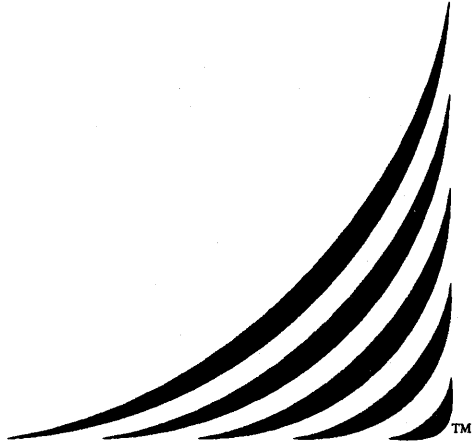
The XIDAK UCR System (bug tracking system) provides simultaneous access to a common data base of UCR's. Each user runs his own copy of the command executive (UCREX). UCREX communicates with the common server process "ucrsrv". Refer to the document (available by request) "MAINSAIL(R) UCR Executive Command Summary" for more information.

G.4. Network Server Status: SRVINF

The SRVINF utility reports on the status of XIDAK servers on the hosts whose names the user supplies. SRVINF prompts for a line of host names and prints a summary of the current connections to the hosts.

G.5. MAINSAIL Kermit

MAINKERMIT, a MAINSAIL implementation of the Kermit file transfer program, is shipped with MAINSAIL systems containing STREAMS. Source code and documentation for it are available on request from XIDAK.



MAINKERMIT User's Guide

24 March 1989

xidak™

12. Overview of MAINKERMIT

MAINKERMIT is a version of the Kermit file transfer utility implemented in the portable programming language MAINSAIL. The Kermit protocol was developed by the Columbia University Center for Computing Activities. Kermit is available for a variety of system for a nominal fee from Columbia and from various user group organizations, such as DECUS and SHARE. MAINKERMIT is available in source form on XIDAK's MAINSAIL Public Tape, which can be obtained from XIDAK, at XIDAK's standard media production fee. MAINKERMIT is also shipped in object form to all MAINSAIL users licensed to use the MAINSAIL STREAMS package.

This manual is intended to be used as a supplement to an authorative reference on Kermit, such as the "KERMIT User Guide" (fifth edition. New York: Columbia University Center for Computing Activities, 1984) or "KERMIT: A File Transfer Protocol" (Bedford, Massachusetts: Digital Press, 1987), both by Frank da Cruz. It is assumed that the reader is familiar with basic Kermit concepts, such as "local" and "remote" Kermits, and is familiar with the use of a "typical" Kermit. Copies of Kermit documenation and sources for other Kermit implementations can be obtained from them by writing to:

```
Kermit Distribution
Columbia University Center for Computing Activities
612 West 115th Street
New York, NY  10025
```

MAINKERMIT has been designed to run interactively. Although no wildcard send is currently allowed, a list of files can be specified to be sent as a batch. Also, because File Attribute packets have been implemented, text and data files can be intermixed when two MAINKERMIT's are talking to each other.

12.1. Version

This version of the "MAINKERMIT User's Guide" is current as of Version 12.10 of MAINSAIL.

Feature	
Local operation	*
Remote operation	Yes
Login scripts	No
Transfer text files	Yes
Transfer data files	Yes
Wildcard send	No
File transfer interruption	No
File name collision avoidance	No
Can time out	*
8th-bit prefixing	Yes
Repeat count prefixing	Yes
Alternate block checks	No
Terminal emulation	*
Communication settings	*
Transmit BREAK	*
Support for dialout modems	No
IBM mainframe communication	Yes
Transaction logging	Yes
Session logging	No
Debug logging	No
Packet logging	Yes
Act as server	Yes
Talk to server	Yes
Advanced server functions	Yes
Local file management	Yes
Command/Init files	Yes
File Attribute packets	Yes
Command macros	No
Raw file transmit	No

* -- Depends on capabilities provided by the operating system.

Table 12-1. MAINKERMIT At a Glance

12.2. File Names and Types

MAINKERMIT was designed to be a highly portable, full-featured Kermit, which can reliably transfer both text and data files. Due to the implementation of File Attribute packets, a mixture of text and data files can be batched and sent with no user intervention.

Currently, very little is done in the way of file name translation. File names are converted to lower case unless literal-names has been set. Nothing is done about converting directories from one form to another, or making sure that the target file name is legal for a given system.

12.3. File Transfer

When running MAINKERMIT in local mode, some information is printed during the course of a file transfer. When a File Header packet is received or sent, the file name is printed. During the transfer, a dot is printed for every five Data packets. Table 12.3-1 lists other information shown.

```
.   Five Data packets
Z   End of file
S   Short packet received
N   NAK received
T   Timeout
C   Checksum Error
```

Table 12.3-1. Information Displayed During File Transfer

The keyboard is not active during the transfer, except for normal operating-system specific interupts, such as CTRL-C or CTRL-T.

12.4. Operation

The MAINKERMIT default prompt is "MAINKERMIT> ", indicating that Kermit is waiting for a command. This prompt can be changed by the user to minimize confusion when switching from one system to another. In general, a command line is a command verb followed by an optional list of arguments, separated by spaces. Commands can be typed in from the keyboard, or taken from a file (see the "TAKE" command, Section 13.10). When

MAINKERMIT is first started, the file "kermit.ini" is looked for on the current directory, and if present, commands are executed from it. Command files can be nested.

MAINKERMIT is slightly different from most other Kermits in that file transfer requests can be queued. The "TEXT", "PTEXT", and "DATA" commands allow the user to enter such a request and specify the file type. The "SEND" or "GET" commands start the transfer of the queue. It is posible to distinguish between text and data files during transfer because File Attribute packets have been implemented. The user must note, however, that few other Kermits have implemented this feature. If the other Kermit does not support File Attribute packets, the file is still sent, but may be treated incorrectly.

MAINKERMIT can also be used in a mode similar to other Kermits, where the file type is set with "SET FILETYPE" and file names are specified in the "SEND" command.

Chapter 13 describe the commands to MAINKERMIT in detail.

13. MAINKERMIT Commands

Table 13-1 lists the current MAINKERMIT commands. All commands and arguments listed can be abbreviated to the shortest unambiguous substring.

Command	Description
SERVER	Enter server mode.
RECEIVE	Receive files.
SEND	Send files.
SET	Set a set option.
EXIT	Exit KERMIT to operating system.
QUIT	Exit KERMIT to MAINSAIL.
TAKE	Take commands from a command file.
CONNECT	Start terminal emulator.
GET	Get files from remote server.
FINISH	Tell remote server to finish.
TEXT	Queue a text file.
PTEXT	Queue a portable text file.
DATA	Queue a data file.
RTEXT	Remotely queue a text file.
RPTEXT	Remotely queue a portable text file.
RDATA	Remotely queue a data file.
REMOTE	Execute a remote server command.
REXECUTE	Remotely execute a Kermit command.
HISTORY	Show the file transaction history.
TYPE	Type a local file on the screen.
COPY	Copy one local text file to another.
DIRECTORY	Display the local directory specified.
RENAME	Rename a local file.
DELETE	Delete a local file.
INVOKE	Invoke a MAINSAIL module.
?	Display a list of commands.

Table 13-1. MAINKERMIT Command list

13.1. The "SEND" Command

```
Syntax: SEND, SEND fn, or SEND fn1 fn2
```

The SEND command with no file name specified starts transmitting the current transfer queue, as specified by previous "TEXT", "PTEXT", or "DATA" commands.

If a file name "fn" is specified, the named file is transfered under the current default file type.

If both an input file name "fn1" and an output file name "fn2" are supplied, the file "fn1" is transfered under the name "fn2".

The corresponding Kermit must be in either server or receive mode.

13.2. The "RECEIVE" Command

```
Syntax: RECEIVE
```

The "RECEIVE" command accepts file transfers from a single "SEND" command, then returns to the Kermit command level. If the sending Kermit supports file attributes, the file type is set by the File Attribute packet. If no attribute packet is sent, the file type is set to the default file type.

13.3. The "GET" Command

```
Syntax: GET, GET rfn, or GET rfn1 fn2
```

Request the remote Kermit to send the remote file named. If the remote Kermit is MAINKERMIT, the rest of the remote transfer queue is sent as well. If the output file name "fn2" is specified, the file is stored under that name. The remote Kermit must be in server mode to use this command.

The syntax of this command is different from that for most Kermits. Usually, if no file name is given, the user is prompted for a remote name, then a local file name. In MAINKERMIT a "GET" with no file name requests the remote Kermit to send its transfer queue. This queue can be built using the "RTEXT", "RPTEXT", and "RDATA" commands (see Section 13.6).

13.4. The "TEXT", "PTEXT", and "DATA" Commands

```
Syntax: TEXT fn or TEXT fn1 fn2,
        PTEXT fn or PTEXT fn1 fn2,
        DATA fn or DATA fn1 fn2
```

The "TEXT" command adds the filename "fn" to the local transfer queue as a text file. If the output file name "fn2" is specified, the file "fn1" is transfered under the name "fn2".

The "PTEXT" command works the same as the "TEXT" command except that it does not filter out nulls or other special characters. Character set and eol translation is still done, however. This command should be used for transferring MAINSAIL intmods.

Similarly, the "DATA" command add the filename to the transfer queue as a data file. If the remote Kermit does not support File Attribute packets, it stores the files as its own default value. Currently, no warning is issued in such a situation.

13.5. The "SERVER" Command

```
Syntax: SERVER
```

The "SERVER" command puts Kermit in "server mode". All commands from that point must be in the form of server packets. Table 13.5-1 lists the commands to which the server responds.

13.6. The "RTEXT", "RPTEXT", and "RDATA" Commands

```
Syntax: RTEXT rfn or RTEXT rfn1 fn2,
        RPTEXT rfn or RPTEXT rfn1 fn2,
        RDATA rfn or RDATA rfn1 fn2
```

The "RTEXT" command sends a "TEXT rfn" command to a remote server, which causes the remote filename specified to be added to the remote transfer queue. If the output file name "fn2" is specicied, the file "rfn1" is transferred under the name "fn2".

The "RPTEXT" command acts the same way for a portable text file, and the "RDATA" command acts the same way for a data file. The remote transfer queue can be retrieved by using the "GET" command. These commands work only for a MAINKERMIT server.

Command	Server Response
GET	Send files
SEND	Receive files
FINISH	Exit server mode
REMOTE DIRECTORY	Send directory listing
REMOTE COPY	Copy files
REMOTE TYPE	Send file to display
REMOTE RENAME	Rename a file
REMOTE DELETE	Delete a file
RTEXT	Add "TEXT" file to transfer queue
RPTEXT	Add "PTEXT" file to transfer queue
RDATA	Add "DATA" file to transfer queue
REXECUTE	Execute a Kermit command

Table 13.5-1. Server Commands Supported by MAINKERMIT

13.7. The "REMOTE", "FINISH", and "REXECUTE" Commands

MAINKERMIT may request a number of services from a remote server. In addition to the "SEND", "GET", "RTEXT", "RPTEXT", and "RDATA" commands already described, Table 13.7-1 lists other commands.

Command	Description
REMOTE DIRECTORY {dn}	Request a remote directory be sent. Use the directory name "dn" if supplied.
REMOTE COPY rf1 rf2	Copy remote file "rf1" to "rf2".
REMOTE RENAME rf1 rf2	Rename remote file "rf1" to "rf2".
REMOTE DELETE rf	Delete remote file "rf".
REMOTE TYPE rf	Send the remote file "rf" to the screen.
FINISH	Cause the remote server to exit server mode.
REXECUTE s	Execute the Kermit command "s".

Table 13.7-1. Remote Commands

A few words about the "REXECUTE" command: this causes the string "s" to be sent to the remote Kermit's command exec, exactly as if you had typed it in at the command prompt. This feature is supported by the MAINKERMIT server, but may not be supported by other servers. In general, care should be taken with this command, as it is possible to send commands to the remote server that may confuse it (e.g., if you were to change the line while communication was in progress). On the other hand, it can be useful to alter remote "SET" options. The "RTEXT", "RPTEXT", and "RDATA" commands are implemented through the "REXECUTE" command.

13.8. Local File Manipulation Commands

MAINKERMIT allows some local file manipulation. These commands are listed in in Table 13.8-1.

```
DIRECTORY {dn}       Request a directory be sent. Use the
                     directory name "dn", if supplied.
COPY f1 f2           Copy file "f1" to "f2".
RENAME f1 f2         Rename file "f1" to "f2".
DELETE f             Delete file "f".
TYPE f               Send the file "rf" to the screen.
```

Table 13.8-1. MAINKERMIT Local File Commands

13.9. The "SET" Command

```
Syntax: SET option {value}
```

MAINKERMIT has a number of variables and options. The "SET" command allows the user to tailor these for a specific need. Table 13.9-1 lists the set options available. These options are described in detail below.

13.9.1. "SET LINE deviceName"

"SET LINE" sets the communication device. The device name is the operating-system-specific name for the line you plan to use, e.g., "/dev/ttyb" or "_TXA5:". When the "SET LINE" command is given, Kermit is put in "local mode". By default, Kermit is in "remote mode".

LINE	Communication device name.
FILETYPE	Default file type (text, portable text, or data).
TARGET	Target operating system for data files.
BAUD	Baud rate of communication device.
DEBUG	Toggle the debug switch.
PROMPT	Change the Kermit prompt.

Table 13.9-1. MAINKERMIT "SET" Options

After a "SET LINE" command, it is possible that a "SET BAUD" command may need to be issued. Note that the "SET LINE" command is operating-system-dependent, and may not be implemented on all systems.

13.9.2. "SET BAUD [300|1200|2400|4800|9600]"

Set the baud rate of the line specified in the "SET LINE" command to the baud rate selected. The command is supported only if the host operating system allows the baud to be set. A "SET LINE" command must be issued before this command.

13.9.3. "SET FILETYPE [TEXT|PTEXT|DATA]"

Set the default file type to "TEXT", "PTEXT", or "DATA", as specified. File Attribute packets override this setting, but this is the file type used in the absence of such a packet. By default, the file type is "TEXT".

13.9.4. "SET TARGET osName"

This sets the target used in a data transfer to the operating system name specified. osName is one of the accepted XIDAK operating system name abreviations shown in Table I-1. Normally, this command can be dispensed with, as most machines support 8-bit bytes.

13.9.5. "SET PROMPT s"

Set the Kermit command prompt to the specified string s. This can be useful if you are using MAINKERMIT on two different system.

13.9.6. "SET DEBUG"

Toggle the debug switch. With debug on, packets are dumped to the screen as they are received and sent. This may be useful in tracking down a failing connection.

13.10. The "TAKE" Command

```
Syntax: TAKE fn
```

The "TAKE" command instructs MAINKERMIT to execute commands from the filename "fn". This file may also contain "TAKE" commands. The commands executed are echoed. Command files are especially useful in building transfer queues.

An implict "TAKE" is done on the file "kermit.ini" when MAINKERMIT is first invoked.

13.11. The "CONNECT" Command

The "CONNECT" command causes the terminal to go into a terminal emulator, talking to the line specified by the "SET LINE" command. All characters typed are passed through the line, and all characters received are displayed. This is an operating-system-specific command, as some operating systems cannot support this function. The quality of the terminal emulator may vary from system to system as well, depending on the exact nature of the primitives provided. Commands to the local Kermit are prefixed with CTRL-\ and a single character. Table 13.11-1 lists the control codes.

```
c       Close the connection, return to the local Kermit
b       Send a BREAK (not supported on all systems)
0       Send a NULL
CTRL-\  Send a Control-backslash
?       List the control codes accepted
```

Table 13.11-1. MAINKERMIT Emulator Commands

13.12. The "HISTORY" Command

```
Syntax: HISTORY
```

The "HISTORY" command lists all files transferred since MAINKERMIT was started. The command prints whether the file was sent or received, if it was text, portable text, or data, what the name was, what time it was started and what time it was completed. If the transfer was aborted for any reason, that fact is also noted.

14. Program Interface

MAINKERMIT may be invoked from any MAINSAIL program, by including the module declaration shown in Figure 14-1.

```
MODULE kermit (
    INTEGER
    PROCEDURE    $executeKermitCommand
                             (STRING cm;
                              PRODUCES OPTIONAL STRING
                                  msg);

    STRING $prompt;
);
```

Figure 14-1. MAINKERMIT Module Declartion

The procedure $executeKermitCommand takes a string consisting of a single command line for the MAINKERMIT executive. $executeKermitCommand must be invoked once for each command line; you may not issue more than one line per call. The procedure returns a 0 if there were no errors, a 1 if there was an error, and a -1 if Kermit should cease execution (i.e., a "QUIT" command was given). If there is an error, an error message is returned in the string msg. The interface string $prompt is the prompt string with which MAINKERMIT would prompt the user; if it is to be changed, it may be set directly or by passing a "SET PROMPT" command to $executeKermitCommand.

Appendix H. Status of MAINKERMIT Version 2.0

XIDAK does not currently provide standard product support for MAINKERMIT; however, bug fixes and enhancements can be returned to XIDAK for possible incorporation into subsequent releases of MAINKERMIT by writing to XIDAK.

The following items are proposed for inclusion is some future release of MAINKERMIT:

- Increase the number of "SET" options. A number of parameters should be settable, such as the quote character, terminal escape, etc.; currently, they are not.

- Add more logging features, such as a true packet log and an error log.

- Add support for wildcard characters on "SEND", "GET", "TEXT", "PTEXT", and "DATA" commands.

- Add modem dialout support.

- Add MAINSAIL device module support, so that MAINSAIL can transparently reference remote files.

Appendix I. XIDAK Operating System Abreviations Used with the "SET TARGET" Command

Abbreviation	Operating System Name
aeg	Aegis
aos	AOS/VS
cms	VM/SP CMS
emb	EMBOS
ua20	Apollo MC68020/FPA UNIX
uclp	CLIPPER UNIX
udg	ECLIPSE DG/UX
ui38	Intel 80386 UNIX
ui96	Intel 80960 UNIX
uibm	IBM System/370 UNIX
um20	MC68020/MC68881 UNIX
um68	M68000 UNIX
umv	ECLIPSE MV UNIX
upri	PRISM UNIX
urdg	Ridge 32 UNIX
uspa	SPARC UNIX
uvax	VAX-11 UNIX
uw38	Intel 80386/WTL 1167 UNIX
uxa	IBM System/370 Extended Architecture UNIX
vms	VAX/VMS
xcms	VM/XA SP CMS

Table I-1. XIDAK Operating System Abreviations

Index

(service protocol table) 64, 115

< and > in syntax desciptions 1

> in stream names 19

[and] in syntax desciptions 2

_final procedure 46
_init procedure 45

{ and } in syntax desciptions 2

| in syntax desciptions 2

$acceptClient 60
adaptable RPC clients 36
address RPC parameter 26, 27
advanced STREAMS 6, 10
$altErase 109
alterOK bit 51
ary_alloc_status 44
ary_dims 44
ary_first_elem 44
ary_lb1 44
ary_lb2 44
ary_lb3 44
ary_type 44
ary_ub1 44
ary_ub2 44
ary_ub3 44
asynchronous
  execution simulation 76
  interrupt catching 90

backspace 105
basic STREAMS 6
baud, TTY 8, 105
$becomeServer 24
$bindService 60, 74
$block 16, 63, 92, 103
blocking

I/O 12
in RPC calls 24
break, TTY 8, 108
buffer RPC parameter 26, 27

C RPC 40
$canonicalHostName 60
carriage return 99, 106
charadr RPC parameter 26, 27
child process 5, 9, 65, 67
clear pending I/O 8
$clearStream 101
client 58
process 5
RPC 21, 32
client/server rendezvous 74
$closeStream 18
cmdFile and STREAMS 12
connection 58
connserver 45
controlling terminal 104
cooperating
child process 9, 16, 68
child test 110
coroutine
scheduled 75
service 25
coroutines sharing data 76
$cReadStream 100
create bit 51
$createRendezvousName 73
CTRL-C 9, 87, 105
CTRL-D 94, 105
CTRL-Q 105
CTRL-S 105
CTRL-Z 94, 105
$currentDirectory 65
$cWriteStream 100

deadlock, semaphore 80
debugging an RPC module 22
DEFAULTPROTOCOL service protocol table entry 117
delete 105
bit 51, 77
descendants, waiting for 78

$disableInterrupts 87
$discardOutput 109
dispose_array 46
$disposeSemaphore 78
distributed applications 4

echo of TTY 6, 105
editing, line 105
efficiency, RPC 39
$enableInterrupt 89
$enableInterrupts 9, 87
end-of-file
  on stream 94
  on TTY 105
end-of-line
  and stream 99
  and TTY 106
$eofIndicator 109
eol and TTY 106
$eos 93, 103
  on socket 56
  on TTY 105
$erase 109
errMsg and scheduling 76
$error 63, 93, 103
error, STREAMS 16
errorOK bit 19, 25, 51, 73, 79, 102, 108
$executableBootName 67

featherweight process 5
file
  remote 50
  RPC 50
$fillBuffer bit 96
$flow 107
flow control 105
$flushStream 97, 101
full-duplex TTY 6, 104, 111

gateway, stream 128
gensrv installation 115
$getBaudRate 108
$getHosts 60
$getProtocols 60

half-duplex TTY 6, 17, 104, 111
$hisHostName 57
HOSTNAME service protocol table entry 116
$hostName 63
HOSTPROTOCOL service protocol table entry 116

I/O
  clearing 8
  low-level stream 92
  scheduled 75
$image bit 96, 99
$initializeStreams 15
input bit 19, 51, 101
installation, server 115
interactive child 70
interface, RPC module 26
interprocess
  communication 4
  communication stream 11
  procedure calls 21
$interrupt 109
interrupt 9, 87
  character 105
interrupt character, reading 7
$interruptsEnabled 87
intmod, STREAMS 15
IPC stream 11
$isaTty 16, 110
$isHalfDuplex 16
$isLocked 78
$isScheduled 8, 16

keepNul bit 51
Kermit terminal emulator 80
keyboard interrupt 9, 87
$keyboardInterruptExcpt 9, 87
$killServerExcpt 37

$lastError 63
$lastInputError 16, 94
$lastOutputError 16, 94
$line bit 96, 99, 100
line editing 105
$lineErase 109
linefeed 99, 106

$linefeedKey 109
$lock 78
locking data structures 76, 78
log file, server 37
logFile and STREAMS 12
low-level stream I/O 92

MAINKERMIT terminal emulator 80
map of scheduled coroutines 80
memory stream 11, 114
MEMSTR STREAMS module 79, 114
MS_ADDRESSCODE 43
MS_BITSCODE 43
MS_BOOLEANCODE 43
MS_CHARADRCODE 43
ms_exception 46
MS_INTEGERCODE 43
MS_LONGBITSCODE 43
MS_LONGINTEGERCODE 43
MS_LONGREALCODE 43
MS_REALCODE 43
MS_STRINGCODE 43
$msTimeout 77
multiprocessor program 81
multitasking 75
MYHOST service protocol table entry 115
$myHostName 57, 60

$name 16, 78
NET device module 130, 133
NETSTR 65, 128
network
  distributing application across 81
  protocol module 58
newestVersion 36
$newRemoteModule 24
$newSemaphore 78
node, computer 4
$noFlow 107
$noInterrupt 107
  bit 89
non-blocking I/O 12
NTTY device module 12, 104
null character 108

$octet bit 96, 99, 100
octet 95
oldestVersion 36
$openStream 18
$optionalFirstEol 103
output bit 19, 51, 101

$packet bit 96, 99
parallel processing 81
$parent 9, 15, 68, 110
$parseHostServiceName 60
PDF
  data in RPC call 27
  image 26
$pdf bit 51
$poll 92, 103
$port 60, 63
port 58
  in rendezvous 74
  server 25
primary I/O 104
procedure call, interprocess 21
process 5
  communication 21
  control 69
  rendezvous 5, 73
protocol
  for server port 25
  module 58
  version in RPC 32, 33
$protocolName 57
pseudo-terminal 11, 111
PTY stream 11, 111
PTYPRO 10, 65
  STREAMS module 111
  $tty in child 104

$queueCoroutine 77
$quoteNext 109

random bit 51
re-entrant procedure caveat 76
$readStream 95
Remote Procedure Calls 21
remote

file 50
module execution in child 29
module interface 26
$remoteModuleCls 22, 26, 36
$remoteModuleDefaults 26
rendezvous 5, 73, 114
$reprint 109
$reschedule 77
$returnKey 109
RPC 21
buffer 26, 27
efficiency 39
execution in child 29
file 50
for C 40
module interface 26
protocol version 32, 33
server/client 21, 32
RPC module, debugging 22
rpc_clear_jump 46
rpc_register_jump 46
RPCSRV server module 32, 38
RS-232 stream 11

scheduled
coroutine 5, 12, 75
TTY 8
$scheduledCoroutineMap 80
Scheduler 12, 75
scheduling of I/O 12
SCOMAP module 80
$semaphore 78
semaphore 76
deadlock 80
server 58
generic (RPCSRV) 38
installation 115
log file 37
process 5
RPC 21, 32
server.log 37
server/client rendezvous 74
$serverName 57
SERVICE 58
service protocol table entry 117

service 58
  coroutine 25
  protocol table 64, 115
service version, RPC 33
$serviceName 63
$setBaudRate 108
shared data among tasks 76
shell 112
  as child process 10
  starting with PTYPRO 66
  writing 90
simultaneous remote procedure calls 81
site.cmd 117
socket stream 11, 56
SOCPRO 65
  stream module 9
  $tty in child 104
SRVINF module 133
$startOutput 109
$stopOutput 109
$stream 16
stream 5
  clearing 101
  end-of-line 99
  flushing 101
  gateway 128
  memory 114
  opening and closing 18
stream I/O, low-level 92
STREAMS 4
  intmod 15
STRHDR 15
stub, RPC 22
$success 93
synchronizing scheduled coroutines 78
system shell as child process 10
$systemSupportsScheduling 10, 16
$systemSupportsTimeout 11
$systemSupportsTimeouts 16

task
  scheduled 75
  scheduling 12
TDB 133
terminal

echo 105
emulator example 80
interrupt 9, 87
terminal interrupt character, reading 7
$text bit 96, 99, 100
$timedOut 63, 93, 103
$timeout and STREAMS 12
timeout, STREAMS 11, 92
TTY
break 108
echo 105
end-of-file 105
end-of-line 106
file and STREAMS 12
interrupt 9, 87
stream 6, 104
$tty 6, 15, 104
TTY interrupt character, reading 7
ttyRead and ttyWrite and STREAMS 12
TTYSTR STREAMS module 104

UCRSYS 133
$unbindService 60
$unbuffered bit 96, 99, 100
$unbufferedEol 109
$unlock 78
$userID 65

version
$remoteModuleCls field 36
RPC protocol 32, 33

$waitForDescendants 78
$wordErase 109
$writeStream 98
$writeStreamBreak 108, 113
$writeStreamInterrupt 113

XON/XOFF 105

XIDAK, Inc., 530 Oak Grove Avenue, M/S 101, Menlo Park, CA 94025, (415) 324-8745