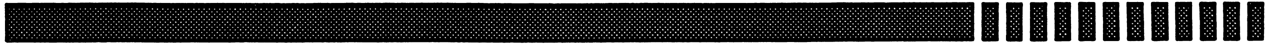
September 1989

Order Number: 311801-002



iPSC®/2 VMSLink

PROGRAMMER'S REFERENCE MANUAL

intel® Corporation

Copyright ©1989 by Intel Scientific Computers, Beaverton, Oregon. All rights reserved. No part of this work may be reproduced or copied in any form or by any means...graphic, electronic, or mechanical including photocopying, taping, or information storage and retrieval systems...without the express written consent of Intel Corporation. The information in this document is subject to change without notice.

Intel Corporation make no warranty of any kind with regard to this material, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Intel Corporation assumes no responsibility for any errors that may appear in this document. Intel Corporation makes no commitment to update or to keep current the information contained in this document.

Intel Corporation assumes no responsibility for the use of any circuitry other than circuitry embodied in an Intel product. No other circuit patent licenses are implied.

Intel software products are copyrighted by and shall remain the property of Intel Corporation. Use, duplication or disclosure is subject to restrictions stated in Intel's software license, or as define in ASPR-7-104.9(a)(9).

The following are trademarks of Intel Corporation and its affiliates and may be used only to identify Intel products:

Above
BITBUS
COMMputer
Concurrent File System
Concurrent Workbench
CREDIT
Data Pipeline
Direct-Connect Module
FASTPATH
GENIUS
I^2ICE
i
im
ICE
iCEL
iCS
iDBP
iDIS
iLBX
Im
iMDDX
iMMX
Insite
$int_e l$
$int_e lBOS$
Intelevision
$int_e ligent$ Identifier
$int_e ligent$ Programming
Intellec
Intellink
iOSP
iPDS
iPSC
iRMX
iSBC
iSBX
iSDM
iSXM
KEPROM
Library Manager
MAP-NET
MCS
Megachassis
MICROMAINFRAME
MULTIBUS
MULTICHANNEL
MULTIMODULE
ONCE
OpenNET
OTP
PC BUBBLE
Plug-A-Bubble
PROMPT
Promware
QueX
QUEST Programming
Quick-Pulse
Ripplemode
RMX/80
RUPI
Seamless
SLD
SugarCube
UPI
VLSiCEL
4-SITE

UNIX is a trademark of AT&T
Excelan is a trademark of Excelan Corporation
EXOS is a trademark or equipment designator of Excelan Corporation
Green Hills Software, C-386, and FORTRAN-386 are trademarks of Green Hills Software, Inc.
Lucid and Lucid Common Lisp are trademarks of Lucid, Inc.
XENIX is a trademark of Microsoft Corporation
VAST2 is a registered trademark of Pacific-Sierra Research Corporation
VP/ix is a trademark of INTERACTIVE Systems Corporation and Phoenix Technologies, Ltd.
NFS is a trademark of Sun Microsystems
Sun Microsystems and the combination of Sun and a numeric suffix are trademarks of Sun Microsystems
VADS and Veridx are registered trademarks of Verdix Corporation
APSO is a service mark of Verdix Corporation
GVAS is a trademark of Verdix Corporation
Ethemet is a registrered trademark of XEROX Corporation
VMS is a trademark of Digital Equipment Corporation

REV.	REVISION HISTORY	DATE
-001	Original Issue	09/89
-002	Revision	10/89

RESTRICTED RIGHTS

Use, duplication or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c) (1) (ii) of the rights in Technical Data and Computer Software clause at 52.227-7013. Intel Corporation, 3065 Bowers Avenue, Santa Clara, California 95051.

PREFACE

PURPOSE/SCOPE

This manual is the Programmer's Reference Manual for the iPSC®/2 VMSLink library routines. The manual assumes that you are an application programmer and are proficient within the VMS environment. The manual contains reference pages that supply syntax information and descriptions of the VMSLink library routines.

ORGANIZATION

This preface contains general information about this manual, applicable documentation, and conventions for argument notation.

Chapter 1 presents general information that describes the relationship between VMS and iPSC/2 systems, describes the differences in using the VMS library routines in the VMS and the UNIX environments, explains how to link VMS applications to the iPSC/2, and shows how to install VMSLink.

Chapter 2 presents syntax, syntax description, argument description, and examples of each library call. Each call also lists possible error messages.

Chapter 3 presents information describing error handling, return codes, and library routine errors.

Appendix A provides a sample communication program between the host and a VMS application to illustrate the use of a few common library calls.

APPLICABLE DOCUMENTS

For additional information, refer to the other members of the iPSC®/2 manual set:

iPSC®/2 User's Guide.
This manual is intended to provide you with enough detail to begin using the iPSC/2 system.

iPSC®/2 System Administrator's Guide.
This manual provides a detailed description of the system administration tasks related to operating and maintaining an iPSC/2 system.

iPSC®/2 C Language Reference Manual.
This manual describes the C compiler for the iPSC/2 system.

iPSC®/2 Fortran Language Reference Manual.
This manual describes the Fortran compiler for the iPSC/2 system.

iPSC®/2 Lisp Programmer's Reference Manual.
This manual describes the iPSC/2 Lisp user interface and the iPSC/2 Lisp concurrent constructs.

iPSC®/2 Lisp Language Reference Manual.
This manual describes the Lisp implementation that runs on the iPSC/2 nodes and its extensions.

iPSC®/2 Programmer's Reference Manual.
This manual describes iPSC/2 commands and iPSC/2 C and Fortran system calls.

iPSC®/2 VX User's Guide.
This manual describes how to develop programs for the iPSC/2-VX vector processing system

UNIX System V Manual Set.
These manuals provide a complete description of the UNIX System V operating system.

MicroVMS and VAX Programming Documentation
These manuals provide a complete description of the MicroVAX/VMS operating system.

NOTATIONAL CONVENTIONS AND SYNTAX

The notation following each argument name in Chapter 2 describes its syntax. The following information translates the character codes used in the argument notation.

Argument notation consists of the following convention:

<name>.<access> <data type>.<passing mechanism>

name Mnemonic that identifies the argument. Argument names for system services are identical to the keywords used to identify the arguments in MACRO programs.

access Describes the type of access that the procedure will (or can) make to the argument:

- *m* Argument is modified, that is, read and written.
- *r* Argument is read only.
- *w* Argument is write only.

data type The following list includes character codes, which indicate the data type of an argument and its associated description:

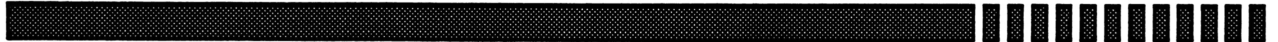
- *l* Longword integer (signed). A 32-bit signed two's complement integer.
- *lu* Longword (unsigned). A 32-bit unsigned quantity.
- *w* Word integer (signed). A 16-bit signed two's complement integer.
- *wu* Word (unsigned). A 16-bit unsigned quantity.
- *z* Unspecified. The calling program has specified no data type. The called procedure should assume the argument is of the correct type.

passing mechanism

Describes the argument passing mechanism that the called routine expects:

d	By descriptor; the contents of the argument list entry is the longword address of a descriptor. The descriptor is two or more longwords that specify further information about the parameter.
r	By reference; the argument list entry is the longword address of the argument.
v	By immediate value; the contents of the argument list entry is itself the argument. Argument list entries for arguments passed by immediate value are always allocated as a longword. Quadword data types can be used as values only for function values, never as arguments.

TABLE OF CONTENTS



CHAPTER 1 GENERAL INFORMATION

INTRODUCTION 1-1

LINKING THE BEST OF TWO SYSTEMS 1-3

SUPPORTING A DISTRIBUTED APPLICATION 1-3

INDUSTRY STANDARD SUPPORT 1-4

VMS and UNIX PROGRAMMING ENVIRONMENT DIFFERENCES 1-6

INSTALLING THE SOFTWARE 1-7

CHAPTER 2 VMS LINK ROUTINES

INTRODUCTION 2-1

CRECV 2-3

CSEND 2-5

FLUSHMSG 2-7

GETCUBE 2-8

INFOCOUNT, INFONODE, INFOPID, INFOTYPE 2-11

IPROBE 2-12

IRECV 2-13
KILLPROC 2-15
LOAD 2-16
MSGDONE 2-18
MSGWAIT 2-19
MYHOST, MYNODE, MYPID 2-20
NODEDIM 2-21
NUMNODES 2-22
RELCUBE 2-23
SETPID 2-24

CHAPTER 3 ERROR INFORMATION

ERROR HANDLING 3-1
RETURN CODE DESCRIPTION 3-2

APPENDIX A iPSC®/2 SAMPLE APPLICATION PROGRAM

APPENDIX B iPSC®/2 TYPES AND TYPESEL MASKS

TYPES B-1
TYPESEL MASKS B-2
Typesel Mask List B-3

LIST OF ILLUSTRATIONS

Figure 1-1. The Software Components of the iPSC® VMSLink Environment........................1-2

Figure 1-2. User and Programmer View of VMSLink..1-4

Figure 1-3. Implementation of VMSLink..1-5

GENERAL INFORMATION 1

INTRODUCTION

VMSLink software allows VMS users to run computationally intensive portions of their applications on the iPSC/2 system while running the rest of their applications under VMS. Application users gain the best of both environments: the familiar user environment of the VAX/VMS minicomputer or workstation, and the supercomputing power of the iPSC/2 system.

VMSLink software allows developers to create applications on a VAX/VMS system (or workstation) which communicate with the iPSC/2. The software consists of an application interface library, two daemons, and the necessary installation software. It uses TCP/IP to communicate with the iPSC/2 and can support any number of VAX/VMS processes (see Figure 1-1).

VMSLink allows you to allocate a cube (**getcube** and **getcubex**), load an application on the cube (**load** and **loadx**), send and receive messages (**csend**, **crecv**, **irecv**, **msgwait**, **msgdone**), get miscellaneous status information and perform housekeeping chores (**relcube**, and **flushmsg**).

The application library sends a special control signal to the System Resource Manager (SRM). Depending on the request, the SRM sends a message to the cube, updates status information, and performs other housekeeping operations. Messages from the cube to the host are captured by the SRM and routed to the remote host where they can be received.

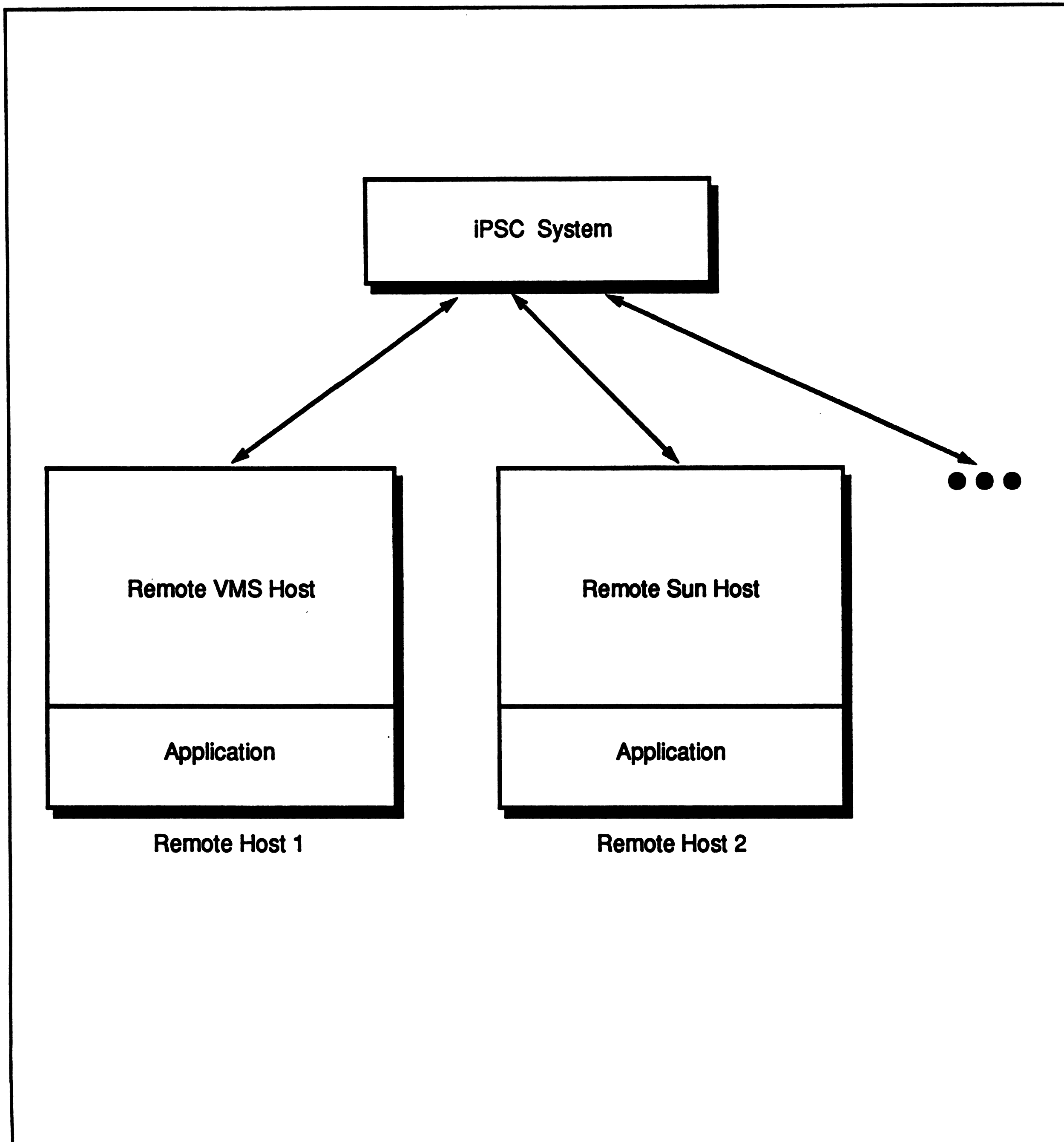


Figure 1-1. The Software Components of the iPSC® VMSLink Environment

LINKING THE BEST OF TWO SYSTEMS

VMSLink software makes message-passing between VMS and iPSC/2 applications transparent to the user. For example, an application involving both interactive graphics and supercomputing can be partitioned into two parts: an interactive front-end running under VMS and a compute engine running on the iPSC/2 system (see Figure 1-2). The applications user is only aware of the local VMS environment and is not concerned with the iPSC/2 connection.

SUPPORTING A DISTRIBUTED APPLICATION

Porting part of a VMS application to the iPSC/2 system is a straightforward process using VMSLink. First, move the compute-intensive portion of the application from the VAX to an iPSC/2 node. Use standard iPSC/2 send and receive message protocols to pass data and commands between the VMS and iPSC/2 portions of the application. VMSLink ensures that the messages are routed correctly between the VAX and the iPSC/2 system. Finally, the iPSC/2 portion can be made to run on multiple iPSC/2 nodes for maximum performance. Meanwhile, the iPSC/2 load module is kept on a local iPSC/2 disk to minimize program load time.

Existing iPSC/2 applications are simple to port to VMS. Often, there is a host-resident portion of the iPSC/2 application already running on the iPSC/2 front-end SRM that can be moved to the VAX. UNIX calls that are not VMS-compatible must be changed in those parts of the application that run under VMS. The host-resident portion can then be recompiled under VMS and linked to the VMSLink library on the VAX. If there is no host-resident portion of the application, then a simple VAX program can be written to load and run the node programs remotely.

Figure 1-2 shows how the VMSLink provides message-passing communications between a VMS-resident user process (left) and one or more iPSC/2 node processes (right). The underlying iPSC/2 compute-intensive processes are seen only by the programmer.

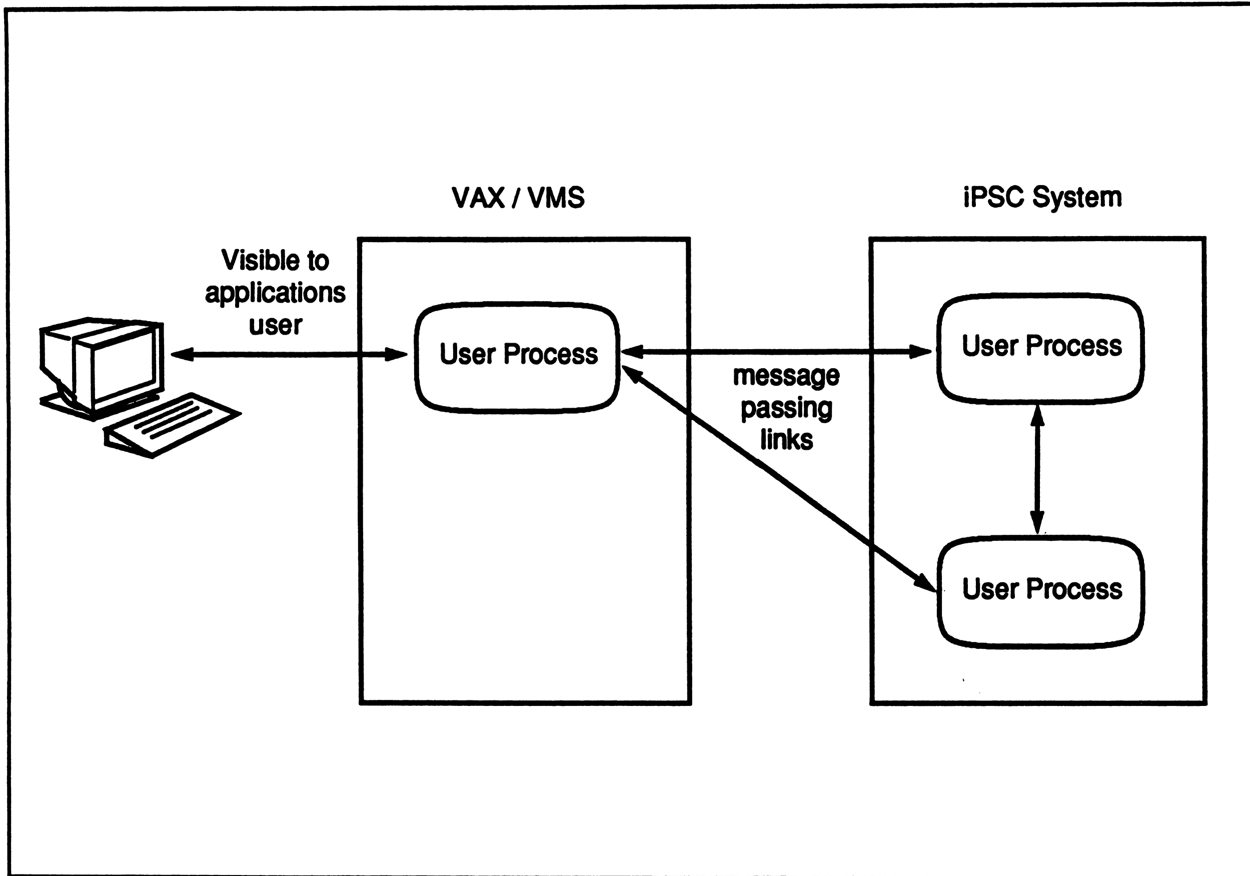


Figure 1-2. User and Programmer View of VMSLink

INDUSTRY STANDARD SUPPORT

VMSLink works with standard software products and communication protocols to make installation simple and applications more portable. Messages passed between the VMS and iPSC/2 processes are sent over a standard Ethernet network connection using TCP/IP protocols and a Berkeley-style sockets interface. DEC's *VMS/Ultrix Connection* software product provides the TCP/IP services under VMS. The VMSLink communications use the same Ethernet as Decnet communications without interference or conflict.

The message-passing library is linked into the VMS process using standard VAX/VMS program development tools. For further convenience, Network File System (NFS) is available to allow remote file access between the VAX and the iPSC/2 system.

Figure 1-3 shows that VMSLink components (shown on right) include iPSC/2 message-passing libraries linked to VAX-resident user processes (left).

The user-transparent server includes processes to relay messages via Ethernet and TCP/IP to and from the iPSC/2 system. User processes for iPSC/2 nodes are typically loaded from a local disk to the iPSC/2 system. Users obtain the *VMS/Ultrix Connection* software (inside left-hand box) from DEC. Ethernet, TCP/IP, and message-passing services are standard on the iPSC/2 system.

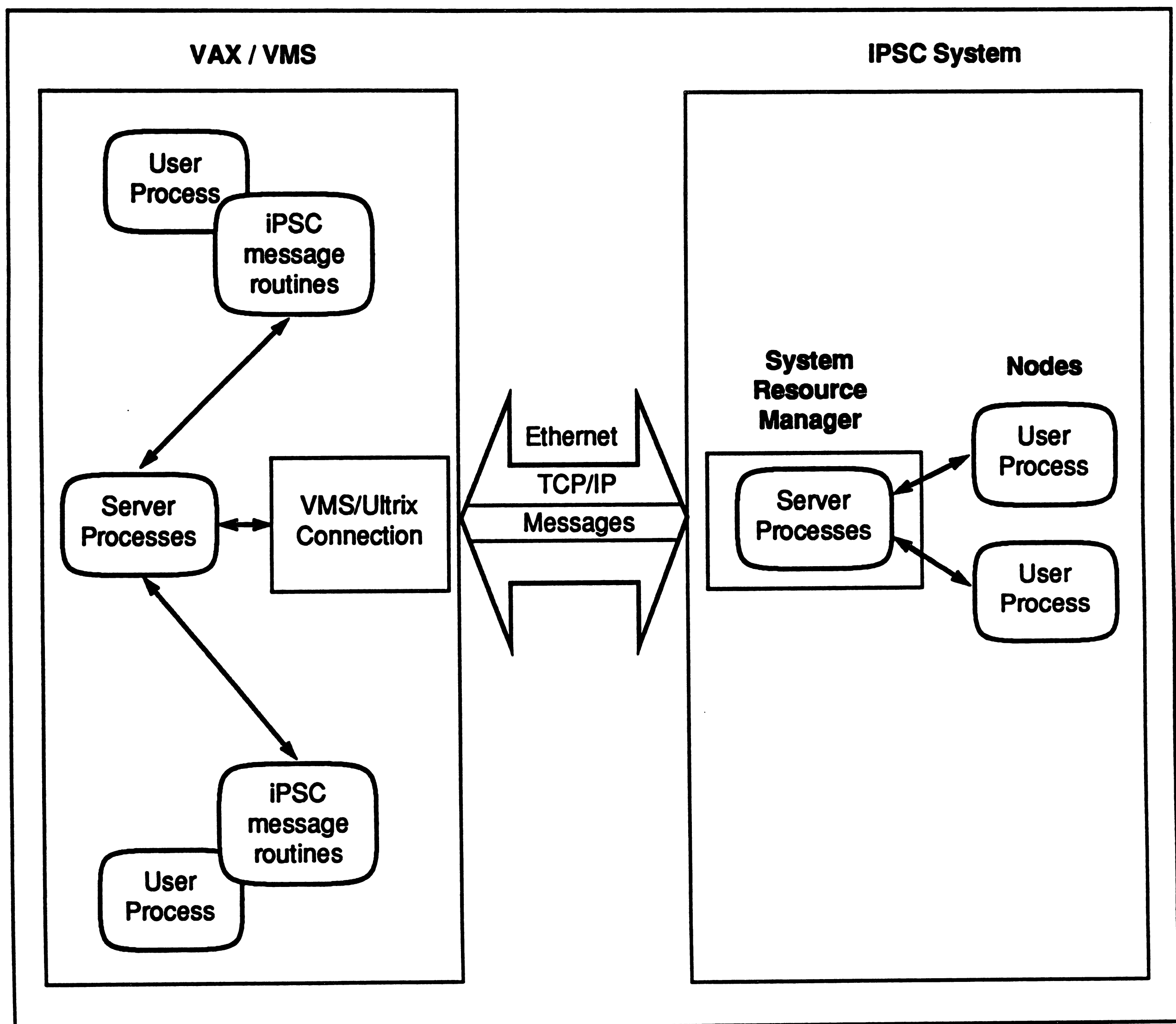


Figure 1-3. Implementation of VMSLink

VMS and UNIX PROGRAMMING ENVIRONMENT DIFFERENCES

Users should note the following differences between the VAX/VMS and UNIX program development environments.

- VAX/VMS (along with the VAX hardware architecture) defines a standard for calling procedures and passing parameters. This allows languages to call procedures from other languages. Information in the *iPSC®/2 Programmer's Reference Manual* concerning different languages *does not* apply to VMS.

- VAX/VMS defines an error handling standard. Consequently, multiple routines (with and without underscores) are not necessary. When encountering an error, the VMSLink software notifies the system, which searches for an error handler. If none is found (or the handler decides not to deal with an error), the default error handler prints a message and continues. For a debugged application, the programmer normally should establish an error handler (with **LIB$ESTABLISH**) to deal with errors in whatever manner is best suited to the application. Failure to provide an error handler results in fatal errors aborting the program.

- VAX/VMS language processors *do not* create executable modules. Rather, they compile source code into object modules. The VAX/VMS LINKER utility performs all linkage operations, including searching appropriate system libraries to satisfy external references.

- Because of the global scope of routines, VAX/VMS programmers should take care to ensure unique global names. The VAX/VMS convention uses the first three letters of a public symbol followed by a dollar sign to relate to the facility name. For example, **BAS$CVT_OUT** is a basic language library routine. For VMSLink, the prefix is "ISC$". All related symbols begin with this prefix.

If you are creating a C language program, you may use:

```
#include <isc.h>
```

to allow use of **CRECV** as well **ISC$CRECV**. This makes the source code more portable to the UNIX environment. Lacking this declaration, you must reference **ISC$CRECV**.

To simplify file maintenance, all files relating to the remote host begin with "ISC" (refer to Appendix A for an example). VAX/VMS conventions dictate the directory for each of these files.

SYS$LIBRARY:ISC.H	C language cube definition header file
SYS$SHARE:ISC_SHR.EXE	The library and support code/data.
SYS$SYSTEM:ISC_D1.EXE	The library to cube daemon.

SYS$SYSTEM:ISC_D2.EXE	The receiver daemon.
SYS$MANAGER:ISC_STARTUP.COM	Command file to install shareable images and start the daemons.

The values for the logical names (e.g., SYS$LIBRARY) are determined at system bootup time by VAX/VMS or the system startup command file (SYS$MANAGER:SYSTARTUP_V5.COM).

INSTALLING THE SOFTWARE

This section gives you instructions to install the VMSLink Software.

NOTE

The iPSC System Software Release 3.1 must be installed prior to installing the VMSLink software. Refer to the *iPSC®/2 System Administrators Guide.*

To install the VMSLink Software Release, perform the following steps on the System Resource Manager (SRM):

1. Login as root.
2. Enter the command:

   installpkg

3. Your monitor displays:

   ```
   Are you entering from tape?
   ```

4. Enter:

   y

5. Your monitor displays:

   ```
   Insert installation tape in drive and press <RETURN>
   ```

6. Insert the VMSLink software tape in the tape drive and press **<RETURN>**.

7. Your monitor displays:

```
Searching tape for list of available packages.

Available packages:

        1. iPSC VMSLink Software Release 1.0

Do you want to install all of the above packages <y/n>
```

8. Enter:

   y

When the VMSLink software is installed, the system prompt returns.

To Install VMSLink on the VAX:

1. Login on the VAX as user SYSTEM.

2. Change to a directory where you will create the VMSLink software. Later, you can delete these files when you have finished using them.

3. Use **ftp** to transfer the **ISC_RELEASE.CO** from the SRM to **isc_release.com** on the VAX.

```
$ ftp
FTP> open <nodename>
.
<login name and password>
.
FTP> cd /usr/ipsc/VMSLink
FTP> get ISC_RELEASE.CO isc_release.com
FTP> quit
```

Where nodename is the SRM.

4. Extract the software from the ISC_RELEASE.COM by entering the following command:

```
$ @isc_release
```

5. To compile and install the software, enter:

```
$ @isc_build
```

- The VMSLink daemons **ISC_D1** and **ISC_D2** start.

- The VMSLink library functions are placed in the default library so you will not need to explicitly link your programs with **ISC_LIB** or **ISC_SHR**.

6. If you need to reboot the VAX, you must also restart the VMSLink daemons by entering the command:

   ```
   $ @sys$manager:isc_startup
   ```

   NOTE

   Place the above command in your system startup command file, **SYS$MANAGER:SYSTARTUP_V5.COM**.

- You can get help for VMSLink functions by entering the following command.

  ```
  $ help isc_routines
  ```

VMSLink is now installed and ready for operation.

VMS LINK ROUTINES 2

INTRODUCTION

This chapter presents descriptions for all VMSLink Library calls. The following calls are supported for both C and Fortran environments:

crecv	Receives a synchronous message; blocks until message received.
csend	Sends a synchronous message; blocks until execution is complete.
flushmsg	Clears the message queue.
getcube	Allocates a set of nodes to run on.
infocount	Gives length of message received after probe or receive operation.
infonode	Gives node ID of node that sent the last received message.
infopid	Gives process ID of the process that sent the last received message.
infotype	Gives type of message received.
iprobe	Determines if a message of a selected type is pending.
irecv	Initiates receipt of an asynchronous message, without waiting for completion.
killproc	Terminates a process on a node.
load	Loads a node process and begins execution.
msgdone	Determines if asynchronous message is complete.

msgwait	Blocks until asynchronous message is complete.
nodedim	Returns the dimension of the allocated cube.
numnodes	Gives the number of nodes allocated in the cube.
relcube	Releases the cube to other users.
setpid	Sets the process ID of a host (VMS) process.

CRECV

Receives a message before proceeding. Program execution is blocked until a matching message arrives.

Synopsis

status.wl.v = **ISC$CRECV** (*typesel.rl.v, rbuf.wz.r, rlen.rl.v*)

Blocks the process while waiting for a message of type *typesel*. Standard message passing wildcards apply (See Appendix B). *rbuf* and *rlen* describe the destination buffer for the message when it is received. Messages too large return an error and the message is lost.

Arguments

typesel.rl.v (passed by value)	Longword containing the type selection mask.
rbuf.wz.r (passed by reference)	Address of the buffer to store the received message.
rlen.rl.v (passed by value)	Longword containing the number of bytes available for use in *rbuf*.

Return Codes

ISC$E_OK	Operation successful.
ISC$E_BADBUFSIZ	Invalid buffer size: *n* bytes.
ISC$E_BUF2SMALL	*n*-byte buffer too small for *n*-byte message.
ISC$E_INVNETCON	Invalid network connection.
ISC$E_IVNUMPAR	Procedure must be called with *n* parameters.
ISC$E_NOATTCUBE	No cube is currently attached.
ISC$E_NOISCPID	A successful call to **SETPID** must precede this call.

CRECV *(cont.)*

CRECV *(cont.)*

ISC$E_NOTRBUF	Buffer is not readable.
ISC$E_SETFRCRCV	Error setting up SRM **irecv**; stat = *n*, errno = *n*.

NOTE

Floating point values are *not* compatible between the SRM and the VAX.

CSEND CSEND

Send a message. Blocks until sending is complete.

Synopsis

status.wl.v = **ISC$CSEND** (*msgtype.rl.v, sbuf.rz.r, slen.rl.v, node.rl.v, pid.rl.v*)

Sends the message described by *sbuf* and *slen* to process *pid* on node of the currently attached cube. The message is of type *msgtype*. When **ISC$CSEND** returns, the message may not have been received yet.

Arguments

msgtype.rl.v (passed by value)	Longword containing the type of the message.
sbuf.rz.r (passed by reference)	Pointer to the message to transmit.
slen.rl.v (passed by value)	Longword containing the number of bytes to transmit.
node.rl.v (passed by value)	Longword giving the node number.
pid.rl.v (passed by value)	Longword giving the process identification.

Return Codes

ISC$E_OK	Operation successful.
ISC$E_BADMSGSIZ	Invalid *n*-byte message. (Maximum allowed is *n*).
ISC$E_INVNETCON	Invalid network connection.
ISC$E_IVNUMPAR	Procedure must be called with *n* parameters.
ISC$E_NOATTCUBE	No cube is currently attached.

CSEND *(cont.)*

CSEND *(cont.)*

ISC$E_NOISCPID	A successful call to **SETPID** must precede this call.
ISC$E_NOTRBUF	Buffer is not readable.

NOTE

Floating point values are *not* compatible between the SRM and the VAX.

FLUSHMSG FLUSHMSG

Flush specified messages from the system.

Synopsis

status.wl.v = **ISC$FLUSHMSG** (*typesel.rl.v, node.rl.v, pid.rl.v*)

Discards messages which have arrived on the remote host, but have not yet been received with a matching *typesel*. In this implementation, *node* and *pid* are ignored; they are effectively -1.

Arguments

typesel.rl.v (passed by value)	Longword containing the type selection mask.
node.rl.v (passed by value)	Longword giving the node number of the destination.
pid.rl.v (passed by value)	Longword giving the process identification on the target node to make the message available.

Return Codes

ISC$E_OK	Operation successful.
ISC$E_IVNUMVPAR	Procedure must be called with *n* to *n* parameters.
ISC$E_NOATTCUBE	No cube is currently attached.
ISC$E_NOISCPID	A successful call to **SETPID** must precede this call.

GETCUBE

Allocate a cube for subsequent use.

Synopsis

status.wl.v = **ISC$GETCUBE** (*cubename.rz.r, cubetype.rz.r, srmname.rz.r, keep.rl.v*)

This call performs the same function as **ISC$GETCUBEX**, except the resulting type is not available.

status.wl.v = **ISC$GETCUBEX** (*cubename.rz.r, cubetype.rz.r, srmname.rz.r, keep.rl.v, result.wz.r*)

This call allocates a cube of type *cubetype* via the SRM *srmname* and assigns it a name of *cubename*. If *keep* is non-zero, the allocation remains after the host program exits. *result* is written with the type of nodes allocated. *result* should be at least 20 bytes long.

Arguments

cubename.rz.r (passed by reference)

Address of a null-terminated ASCII string specifying the name of the subcube if the request is successful. Up to 20 characters may be specified.

cubetype.rz.r (passed by reference)

specifies the size and type of the cube you want to allocate. Specify size first followed by type in a contiguous string (no spaces). Without this option, you get the largest available cube. The number of nodes allocated will always be a power of two. If the size you specify is not a power of two, it will be rounded up.

GETCUBE *(cont.)*

GETCUBE *(cont.)*

Size can be specified in one of two ways:

n	where *n* is the number of nodes (for example: 8) or,
d*n*	where **d** indicates dimension and *n* is the size of the dimension (for example: d3).

Specifying no type gets a cube of any type. Other choices are:

vx	nodes with vector companion board
sx	nodes with SX installed
f	nodes with 387 coprocessor installed
m1	nodes with at least 1M byte of memory each
m4	nodes with at least 4M bytes of memory each
m8	nodes with at least 8M bytes of memory each
m16	nodes with at least 16M bytes of memory each
n*s*	gets a contiguous/adjacent set of nodes starting with slot *s*

srmname.rz.r (passed by reference) — Address of a null-terminated ASCII string specifying the network name of the SRM node.

keep.rl.v (passed by value) — 0 if the cube is to automatically release when the program exits. Otherwise the cube is retained until an explicit call to **ISC$RELCUBE** is performed.

result.wz.r (passed by reference) — Address of a buffer of at least 20 characters in which a null-erminated string will be written describing the actual cube allocated.

GETCUBE *(cont.)*

GETCUBE *(cont.)*

Return Codes

ISC$E_OK	Operation successful.
ISC$E_BADGETCUB	SRM unable to allocate requested cube.
ISC$E_BADHSTNAM	Invalid host name "*hostname*".
ISC$E_INVNETCON	Invalid network connection.
ISC$E_IVNUMPAR	Procedure must be called with *n* parameters.
ISC$E_NOTRBUF	Buffer is not readable.
ISC$E_NOTWBUF	Buffer is not writable.

INFOCOUNT, INFONODE, INFOPID, INFOTYPE

Return information about a pending or received message.

Synopsis

status.wl.v = **ISC$INFOCOUNT ()**

Returns the number of bytes present in the last message received with **CRECV** (**MSGWAIT** successful, **MSGDONE** = 1) or tested with **IPROBE**. This is true even if the message did not fit in a receive buffer.

status.wl.v = **ISC$INFONODE ()**

Returns the node number originating the last message received with **CRECV** (**MSGWAIT** successful, **MSGDONE** = 1) or tested with **IPROBE**.

status.wl.v = **ISC$INFOPID ()**

Returns the *pid* originating the last message received with **CRECV** (**MSGWAIT** successful, **MSGDONE** = 1) or tested with **IPROBE**.

status.wl.v = **ISC$INFOTYPE ()**

Returns the message type of the last message received with **CRECV** (**MSGWAIT** successful, **MSGDONE** = 1) or tested with **IPROBE**.

Arguments

None

Return Codes

See Synopsis

IPROBE

IPROBE

Determine whether a message of a selected type is pending.

Synopsis

status.wl.v = **ISC$IPROBE** (*typesel.rl.v*)

Returns 0 if no message of *typesel* is ready to be received. Returns 1 if one or more messages matching *typesel* are ready. If 1 is returned, the INFO calls can be used to get more information about the first message ready.

Arguments

typesel.rl.v (passed by value)	Longword containing the type selection mask.

Return Codes

0	See synopsis.
1	See synopsis.
ISC$E_IVNUMPAR	Procedure must be called with *n* parameters.
ISC$E_NOATTCUBE	No cube is currently attached.
ISC$E_NOISCPID	A successful call to SETPID must precede this call.

IRECV IRECV

Asynchronously receive a message.

Synopsis

status.wl.v = **ISC$IRECV** (*typesel.rl.v, rbuf.wz.r, rlen.rl.v, msg_id.wl.r*)

Queues a request to receive a message of type *typesel* into buffer *rbuf* of up to *rlen* bytes. If queueing is successful, *msg_id* is written with a number identifying the queued request. This value may be passed to **ISC$MSGWAIT** to complete the pending I/O, or **ISC$MSGDONE** to see if a matching message has arrived.

Arguments

typesel.rl.v (passed by value)	Longword containing the type selection mask.
rbuf.wz.r (passed by reference)	Address of the buffer to store the received message.
rlen.rl.v (passed by value)	Longword containing the number of bytes available for use in *rbuf*.
msg_id.wl.r (passed by reference)	Longword in which a message identification is written.

Return Codes

ISC$E_OK	Operation successful.
ISC$E_BADBUFSIZ	Invalid buffer size: *n* bytes.
ISC$E_INVNETCON	Invalid network connection.
ISC$E_IVNUMPAR	Procedure must be called with *n* parameters.
ISC$E_NOATTCUBE	No cube is currently attached.
ISC$E_NOISCPID	A successful call to **SETPID** must precede this call.

IRECV *(cont.)*

IRECV *(cont.)*

ISC$E_NOTWBUF	Buffer is not writable.
ISC$E_SETFRCRCV	Error setting up SRM irecv; stat = *n*, errno = *n*.

NOTE

Floating point values are *not* compatible between the SRM and the VAX.

KILLPROC KILLPROC

Terminate a process.

Synopsis

status.wl.v = **ISC$KILLPROC** (*node.rl.v, pid.rl.v*)

Terminates a process running on a node.

Arguments

node.rl.v (passed by value)
: Longword containing the node selection mask.

pid.rl.v (passed by value)
: An integer value that specifies the ID of the process that you want to terminate. *Pid* must match a *pid* given for a previous load, or it may be -1 to terminate all processes. Valid *pids* include any integer value, but negative numbers other than -1 are reserved for system programs.

Return Codes

ISC$E_OK	Operation successful.
ISC$E_INVNETCON	Invalid network connection.
ISC$E_IVNUMPAR	Procedure must be called with *n* parameters.
ISC$E_NOATTCUBE	No cube is currently attached.
ISC$E_NOISCPID	A successful call to SETPID must precede this call.
ISC$E_SETFRCRCV	Error setting up SRM irecv; stat = *n*, errno = *n*.

LOAD LOAD

Load a node process.

Synopsis

status.wl.v = **ISC$LOAD** (*filename.rz.r, node.rl.v, pid.rl.v*)

Equivalent to **ISC$LOADX** with *FLAGS* = 1, *argv* describing the SRM filename as parameter zero, and *nodearr* is a single entry consisting of *node* (that is, *NUM_NODES* = 1). Note that *filename* must begin with "SRM:" in release v1.0.

status.wl.v = **ISC$LOADX** (*filename.rz.r, nodearr.rla.r, num_nodes.rl.v, pid.rl.v, flags.rl.v, argv.rz.r*)

Loads *filename* onto the nodes specified by *nodearr* and *num_nodes*. **LOAD** assigns the running programs an identification of *pid*. *filename* must be a host file name unless the first 4 characters of *file_name* are "SRM:", in which case the remaining characters must specify an SRM file. *nodearr* is an array of *num_nodes* integers specifying which nodes in the cube to load; -1 is a permissible entry. *argv* provides the parameters to the running process.

Arguments

filename.rz.r (passed by reference)	Address of the null-terminated string specifying the executable node file to load.
nodearr.rla.r (passed by reference)	Address of an array of longwords describing what nodes to load.
num_nodes.rl.v (passed by value)	Longword giving the number of longword entries to use in *nodearr*.
pid.rl.v (passed by value)	Longword providing the process identification of the process started on each node.

LOAD *(cont.)*

LOAD *(cont.)*

flags.rl.v (passed by value)	Load option flag bits. If bit 0 is set, the program runs after the **load** command. A set bit 1 forces a VX node type of load. A set bit 2 loads an unprotected process for diagnostic purposes. Bits 3 through 33 are reserved and must be 0.
argv.ral.r (passed by reference)	Address of an array of pointers. Each pointer in the array gives the address of a null-terminated string which represent one argument. The array is terminated by a null pointer.

Return Codes

ISC$E_OK	Operation successful.
ISC$E_BADNUMNOD	Number of nodes must be between *n* and *n*.
ISC$E_INVNETCON	Invalid network connection.
ISC$E_IVNUMPAR	Procedure must be called with *n* parameters.
ISC$E_NOATTCUBE	No cube is currently attached.
ISC$E_NOISCPID	A successful call to **SETPID** must precede this call.
ISC$E_NOTRBUF	Buffer is not readable.
ISC$E_NOTSRMFIL	Invalid load file; name must begin with "SRM:".
ISC$E_SETFRCRCV	Error setting up SRM **irecv**; stat = *n*, errno = *n*.

MSGDONE MSGDONE

Determine whether an asynchronous send or receive operation is complete.

Synopsis

status.wl.v = **ISC$MSGDONE** (*msg_id.rl.v*)

Returns 1 if the message finishes I/O. 0 indicates I/O is incomplete. Other values indicate an error. If 1 is returned, *msg_id* is subsequently invalid and the **ISC$INFO...** calls are defined for this message.

Arguments

msg_id.rl.v (passed by value)	Longword containing the message ID returned from a previous call to **ISC$IRECV**.

Return Codes

0	See synopsis.
1	See synopsis.
ISC$E_BUF2SMALL	*n*-byte buffer too small for *n*-byte message.
ISC$E_INVNETCON	Invalid network connection.
ISC$E_IVMSGID	Message identification is not from an active **IRECV**.
ISC$E_IVNUMPAR	Procedure must be called with *n* parameters.
ISC$E_NOATTCUBE	No cube is currently attached.
ISC$E_NOISCPID	A successful call to **SETPID** must precede this call.

MSGWAIT

MSGWAIT

Wait for completion of an asynchronous receive operation.

Synopsis

status.wl.v = **ISC$MSGWAIT** (*msg_id.rl.v*)

Returns when the message identified by *msg_id* (returned from **IRECV**) has finished I/O. The value given for *msg_id* is subsequently invalid.

Arguments

msg_id.rl.v (passed by value)	Longword containing the message identification returned from **ISC$IRECV**.

Return Codes

ISC$E_OK	Operation successful.
ISC$E_BUF2SMALL	*n*-byte buffer too small for *n*-byte message.
ISC$E_INVNETCON	Invalid network connection.
ISC$E_IVMSGID	Message identification is not from an active **IRECV**.
ISC$E_IVNUMPAR	Procedure must be called with *n* parameters.
ISC$E_NOATTCUBE	No cube is currently attached.
ISC$E_NOISCPID	A successful call to **SETPID** must precede this call.

MYHOST, MYNODE, MYPID

MYHOST, MYNODE, MYPID

Obtain the ID of the host, node, and pid.

Synopsis

status.wl.v = **ISC$MYHOST ()**

Returns the pid *of* the host for the currently attached cube. If no cube is currently attached, or a *pid* has not been set, this returns -1.

status.wl.v = **ISC$MYNODE ()**

Returns the node *id* of the host for the currently attached cube. This is the same as **myhost()** on the host.

status.wl.v = **ISC$MYPID ()**

Returns the *pid* last set with **setpid**. If no *pid* has been set, the result is -1. **MYPID** can be used to re-establish or test if a previous **getcube** (with a *keep* = 1) is used in this process (or a program in the same operating system process).

Arguments

None

Return Codes

None

NODEDIM

Returns the dimension of the allocated cube.

Synopsis

status.wl.v = **ISC$NODEDIM ()**

Returns the dimension of the currently attached cube. This is a number from **0** to **7** which corresponds to cube sizes of **1** to **128**. If no cube is currently attached, **-1** is returned.

Arguments

None

Return Codes

See Synopsis

NUMNODES

NUMNODES

Obtain the number of nodes in the cube.

Synopsis

status.wl.v = **ISC$NUMNODES ()**

Returns the number of nodes in the currently attached cube. This is a number from 0 to 128. If no cube is currently attached, -1 is returned.

Arguments

None

Return Codes

See Synopsis

RELCUBE

RELCUBE

Release a cube.

Synopsis

status.wl.v = **ISC$RELCUBE ()**

Deallocates any cube currently obtained with **ISC$GETCUBE**.

Arguments

None

Return Codes

ISC$E_OK	Operation successful.
ISC$E_INVNETCON	Invalid network connection.
ISC$E_IVNUMPAR	Procedure must be called with *n* parameters.
ISC$E_NOATTCUBE	No cube is currently attached.

SETPID SETPID

Sets the process ID of a host process.

Synopsis

status.wl.v = **ISC$SETPID** (pid.rl.v)

Sets identification of a host program. Subsequent message passing refers to *pid* as the current program. Therefore, you must do this call before using the message passing calls.

Arguments

pid.rl.v (passed by value)	Longword containing the value to set this process as an identification to passing messages to and from the cube.

Return Codes

ISC$E_OK	Operation successful.
ISC$E_ISCPIDSET	ISC *pid* is already set (SETPID already called).
ISC$E_ISCPIDUSE	ISC *pid* is already in use on this node.
ISC$E_IVISCPID	ISC *pid* must be non-negative.

ERROR INFORMATION 3

This chapter provides information describing return codes and errors when running VMSLink library routines.

ERROR HANDLING

The standard VAX/VMS error handling mechanism is used to report errors. By default, severe errors (severity = *fatal*) are signaled, results in an informative message, and terminates the program. Less severe errors (severity = *warning* or *error*) are not signaled and return an even-valued status code. Successful calls (severity = *informational* or *success*) return an odd value.

An error handler is established via the **LIB$ESTABLISH** library call (or "MOVAL <#exception handler>, (AP)" for assembly language programmers). Refer to the chapter on Run-Time Errors in the *VAX/VMS Programmer's Manual* for information on writing a condition handler.

Error codes are direct universal symbols available to all programs on a remote host. Use the prefix "ISC$E_" in front of the mnemonic. For example:

```
stat = isc$setpid (9);
if (stat == &ISC$E_ISCPIDSET)
    ...                                   /* action to take if pid already set */
```

This tests to see if **SETPID** returned *ISCPIDSET*.

Since the symbol is a direct value (rather than the address of a location containing the value), special treatment is needed in most high-level languages. In C, the address-of-operator (&) is used. In FORTRAN, %loc (*ISC$E_ISCPIDSET*) is used. In assembly, immediate addressing is used: ***CMPL R0, #ISC$E_ISCPIDSET***.

RETURN CODE DESCRIPTION

Each routine in Chapter 2 lists return codes and a brief description for each call. This section alphabetically lists each return code and provides detailed information.

Except for the **ISC$INFO...** and **ISC$MY...** calls, an odd value indicates successful completion, even values indicate a failure of some kind. The code specifies the exact error.

The **ISC$INFO...** calls give no error indication. It is the caller's responsibility to ensure that the value returned is meaningful only after a successful **CRECV**, **IPROBE**, **MSGWAIT**, or **MSGDONE**.

ISC$E_OK

Operation successful.

ISC$E_BUF2SMALL

n-byte buffer too small for *n*-byte message.

Severity: fatal

A message bigger than the receive buffer arrived. Use a buffer of at least this size.

ISC$E_BADBUFSIZ

Invalid buffer size: *n* bytes.

Severity: fatal

The buffer passed to **ISC$IRECV** or **ISC$CRECV** must be at least one byte in length.

ISC$E_BADGETCUB

SRM unable to allocate requested cube.

Severity: error

The SRM could not allocate the requested cube. This is probably caused by insufficient nodes of the specified type being presently available in a required configuration.

ISC$E_BADHSTNAM

Invalid host name "*hostname*".

Severity: fatal

The host name (given by the *srmname* parameter) is not a known network node. Check the name and spelling. If correct, the name may be missing from the local node's host database.

ISC$E_BADMSGSIZ

Invalid *n*-byte message. (Maximum allowed is *n*).

Severity: fatal

The remote host as well as the SRM do not support sending a message longer than a predetermined limit set by the version of the remote host transport layer.

ISC$E_BADNUMNOD

Number of nodes must be between *n* and *n*.

Severity: fatal

The *num_nodes* parameter to **ISC$LOADX** must be within a predetermined architectural limit of the cube. Note that one is the most common value for this parameter, especially when used with the first node array element containing -1.

ISC$E_INVNETCON

Invalid network connection.

Severity: fatal

A network error occurred attempting to establish a network connection to the SRM or an existing connection has failed. In either case, the cube should be considered released and unusable.

ISC$E_IVISCPID

ISC *pid* must be non-negative.

Severity: fatal

The argument to **ISC$SETPID** must be greater than or equal to zero. Negative values are reserved for internal system use and future assignment.

ISC$E_IVMSGID

Message identification is not from an active **IRECV**.

Severity: fatal

The argument to **MSGDONE** or **MSGWAIT** is not a valid message identification. The message may not be a valid ID because the identification code did not come from **ISC$IRECV**. Or, a previous call to **ISC$MSGDONE** (or **ISC$MSGWAIT**) returned the message, which would invalidate the message ID.

ISC$E_IVNUMPAR

Procedure must be called with *n* parameters.

Severity: fatal

The function was called with fewer or greater parameters than expected. Correct the corresponding source code.

ISC$E_IVNUMVPAR

Procedure must be called with *n* to *n* parameters.

Severity: fatal

The function was called with a number of parameters greater than or less than the acceptable range. Correct the offending source code.

ISC$E_ISCPIDSET

ISC *pid* is already set (**SETPID** already called).

Severity: error

A *pid* has already been set for this process. The *pid* cannot be changed unless the cube is first released.

ISC$E_ISCPIDUSE

ISC *pid* is already in use on this node.

Severity: error

The *pid* passed to **SETPID** is already in use by another process on this node.

ISC$E_NOATTCUBE

No cube is currently attached.

Severity: fatal

A successful call to **ISC$GETCUBE** or **ISC$GETCUBEX** is required before invoking this call.

ISC$E_NOISCPID

A successful call to **SETPID** must precede this call.

Severity: fatal

A successful call to **ISC$SETPID** is required before invoking this call.

ISC$E_NOTRBUF

Buffer is not readable.

Severity: fatal

A parameter points to an object that is not readable. This condition is normally an access violation. The most common cause of this error is passing a null pointer.

ISC$E_NOTSRMFIL

Invalid load file; name must begin with "SRM:".

Severity: fatal

The **ISC$LOAD** or **ISC$LOADX** function does not currently support loading a file from the remote host. You must specify a load file residing on the currently attached SRM. The notation to indicate this is "SRM:<*srm_file_name*>". For example, to load the file **"app"** from directory **"/usr/ipsc"** onto all nodes and assign the *pid* of zero, use:

ISC$LOAD ("SRM:/usr/ipsc/app", -1, 0)

Do not confuse this filename notation with the remote file copy syntax used between networked UNIX nodes; the **ISC$GETCUBE** routine (which must have been previously successful) determines which node is attached. The "SRM:" in the filename string indicates the load file is to reside on the SRM. Future releases of the remote host might allow the load file to reside on the remote.

ISC$E_NOTWBUF

Buffer is not writable.

Severity: fatal

A parameter points to an object that is not writable (it could also be non-readable as well). This condition is normally an access violation. The most common cause of this error is passing a null pointer.

ISC$E_SETFRCRCV

Error setting up SRM **IRECV**; stat = *n*, errno = *n*.

Severity: fatal

The SRM was unable to issue an **IRECV** for the specified *force* type message. This could be caused by too many outstanding *force* type receives. The SRM does this on behalf of the remote user request to insure *force* type messages are captured for routing to the remote. The SRM always has an **IRECV(-1)** pending on behalf of the remote host. This will not capture a *force* type.

iPSC®/2 SAMPLE APPLICATION PROGRAM A

This example consists of two programs to demonstrate VMSLink operation. The first program runs on the cube. It formats a message describing the message received from the VMSLink and sends it back for viewing.

```
main ()
{       int     light = 0;                      /* light describes the node board's*/
                                                /* current led state. 0 = off. */
        int     do_msg (), millisleep ();

        for (;;) {
                if (iprobe (-1L))               /* if any message pending */
                        do_msg (-1L);           /* process it */

                led (light = !light);           /* make light blink */
                millisleep (30);                /* slow node down enough so humans*/
                                                /* can see the led blink.*/
        }
}

do_msg (type)
long    type;
{
        int     stat, src_node, src_pid, src_count, src_type;
        unsigned char buf [256*1024];           /* maximum-length buffer */
```

```
                                                    /* receive pending message */
        stat = _crecv (type, buf, sizeof (buf));
        src_node = infonode ();                     /* get node number it's from */
        src_pid = infopid ();                       /* pid of sender */
        src_count = infocount ();                   /* number of bytes in message */
        src_type = infotype ();                     /* type of message */
        if (stat < 0)
                sprintf (buf, "Error %d from _crecv", stat);
        else
                sprintf (buf,
                "Received %d-byte message from pid %x on node %d of type %d",
                        src_count, src_pid, src_node, src_type);
        csend (1, buf, strlen (buf)+1, src_node, src_pid);
}

                        /* Millisleep is like sleep, but uses the cube library call MCLOCK*/
                        /* to accurately time sub-second periods. */
millisleep (msec)
        int     msec;
{
        int     start = mclock ();

        start = mclock ();
        while (mclock() - start < msec)             /* while time has not expired... */
                flick ();                           /* give up the processor, if possible */
}
```

Programming example of VMSLink. This program runs on the VMS remote host, it communicates with the program above.

```
#include <isc.h>

int
main (int argc, char **argv)
{
        int     stat, j;
        char    name [20], *nodename, *nodenum;

        printf ("Starting\n");

        if (argc != 2) {                            /* if not enough parameters */
        printf ("usage:\n%s srm-node-name\n", argv [0]);
        return 4;                                   /* return FATAL error code */
}
        nodename = argv [1];
        nodenum = "1";
```

```
	stat = getcubex (	"democubename",		/* cubename */
				nodenum,		/* type of cube */
				nodename,		/* srmname */
				0,			/* no keep after prog exit */
				name);			/* resulting name type */
	if (stat != E_OK) {
		printf ("Getcube failed, status = %x\n", stat);
		exit (stat);
	}

	printf ("Getcube (srm: %s) successful: cube type is '%s'\n",
		nodename, name);

	setpid (0);

	stat = load ("SRM:/tmp/demo", /*node*/ -1, /*pid*/0);
	if (stat != E_OK) {
		printf ("Cube load failed, status = %x\n", stat);
		exit (stat);
	}

	for (j = 1; j <= 10; j++) {			/* send/recv ten messages */
		char	*sbuf = "This is some test data to send";

	stat = csend (	j,				/* type */
		sbuf,					/* buffer */
		j*3,					/* number of bytes */
		-1,					/* destination node */
		0);					/* destination pid */

		if (!(stat & 1))			/* error if status even */
			printf ("Error from csend: %x\n", stat);
		getmsg ();
	}
	relcube ();
}

getmsg ()
{
	int		stat, len;
	unsigned char	buf [256*1024];		/* buffer for max length msg */

	stat = crecv (-1, buf, sizeof(buf));
	printf ("reply from node %d, pid %d, len %d, type %d:\n",
		infonode (), infopid (), infocount(), infotype());
	printf ("'%s'\n", buf);
}
```

The output from the interaction between these two programs is as follows:

```
Starting
Getcube (srm: hv) successful: cube type is '1m8n0'
reply from node 0, pid 0, len 55, type 1:
'Received 3-byte message from pid 0 on node 1 of type 1'
reply from node 0, pid 0, len 55, type 1:
'Received 6-byte message from pid 0 on node 1 of type 2'
reply from node 0, pid 0, len 55, type 1:
'Received 9-byte message from pid 0 on node 1 of type 3'
reply from node 0, pid 0, len 56, type 1:
'Received 12-byte message from pid 0 on node 1 of type 4'
reply from node 0, pid 0, len 56, type 1:
'Received 15-byte message from pid 0 on node 1 of type 5'
reply from node 0, pid 0, len 56, type 1:
'Received 18-byte message from pid 0 on node 1 of type 6'
reply from node 0, pid 0, len 56, type 1:
'Received 21-byte message from pid 0 on node 1 of type 7'
reply from node 0, pid 0, len 56, type 1:
'Received 24-byte message from pid 0 on node 1 of type 8'
reply from node 0, pid 0, len 56, type 1:
'Received 27-byte message from pid 0 on node 1 of type 9'
reply from node 0, pid 0, len 57, type 1:
'Received 30-byte message from pid 0 on node 1 of type 10'
```

iPSC®/2 TYPES AND TYPESEL MASKS

TYPES

The *type* parameter used in message passing calls is a user-defined variable typically used to identify the kind of information contained in the message being sent. Types 0 to 999,999,999 are normal types that can be used by any send or receive call. **Note:** Types 1,000,000,000 to 1,073,741,823 and 2,000,000,000 and up are used by the system and should be avoided. Their use may produce unpredictable results. Types 1,073,741,824 to 1,999,999,999 are special force types intended specifically for the **sendrecv** calls. Force types have three special properties:

1. A message with a force type bypasses the normal flow control mechanisms and is not delayed by clogged message buffers on the node.

2. Force types do not match the -1 wildcard type selector. This property can be used to guarantee that the message is received by the proper buffer, no matter what other messages are also received.

3. A message with a force type is discarded if no receive is posted (as when the receiving process has been killed). In general, bypassing the normal flow control mechanisms causes no problem because the **sendrecv** calls guarantee that a receive is posted for the message.

TYPESEL MASKS

The *typesel* parameter used in receive calls is an integer value that specifies the type(s) of message you are waiting for in a **probe, receive,** or **flush** operation. You assign a *type* to a message when you initiate a send operation. *Typesel* (type selector) allows you to select a specific message type or a set of message types based on a 32-bit mask. *Typesel* can be set as follows:

- If *typesel* is a non-negative integer, a specific message type will be recognized. All other messages will be ignored.
- If *typesel* is **-1**, the first message to arrive for the process that initiated a **probe** or **receive** operation will be recognized. After the first message has been received, you can use **-1** again to receive or probe the next message, and so on.
- If *typesel* is any negative number other than **-1**, a set of message types will be recognized. In this case, bits 0-29 of *typesel* correspond to types 0-29. For example, if bit number 3 is set to 1 in the *typesel*, then a message of type 3 will be recognized. If bit number 3 is set to 0, then a message of type 3 will be ignored.

  Bit 30 allows you to select all types greater than 29 as a group. Bit 30 can be used in conjunction with bits 0-29, as desired. Bit 31 set to 1 makes the *typesel* parameter negative and indicates that it is a mask.

To generate a mask, add the hexadecimal numbers associated with the *types* you want to select to the constant, **0x80000000** (See Table B-1). For example, if you want to receive message types 1, 2, 5, and 12, add the following hex numbers:

```
0x2, 0x4, 0x20, 0x1000 + 0x80000000 = 0x80001026
```

then enter

```
crecv (0x80001026, buf, len);
```

Or, if you want to receive any message except type 0 use:

```
crecv (0xFFFFFFFE, buf, len);
```

Typesel Mask List

The following list gives the hexadecimal number associated with bits 0-31:

Type	Hex Number
0	0x00000001
1	0x00000002
2	0x00000004
3	0x00000008
4	0x00000010
5	0x00000020
6	0x00000040
7	0x00000080
8	0x00000100
9	0x00000200
10	0x00000400
11	0x00000800
12	0x00001000
13	0x00002000
14	0x00004000
15	0x00008000
16	0x00010000
17	0x00020000
18	0x00040000
19	0x00080000
20	0x00100000
21	0x00200000
22	0x00400000
23	0x00800000
24	0x01000000
25	0x02000000
26	0x04000000
27	0x08000000
28	0x10000000
29	0x20000000
Other types	0x40000000