



**iRMX™ 86 HUMAN INTERFACE  
REFERENCE MANUAL**

---

**iRMX™  
86  
OPERATING  
SYSTEM**



# **iRMX™ 86 HUMAN INTERFACE REFERENCE MANUAL**

Order Number: 9803202-03

REV.	REVISION HISTORY	PRINT DATE
-01	Original Issue.	5/81
-02	Adds information about the Disk Verification Utility, the BACKUP and RESTORE commands, and exceptional condition codes; changes information about the SUBMIT and FORMAT commands; corrects technical and typographical errors; and documents Release 4 of the iRMX 86 Operating System.	11/81
-03	Removes information about commands (iRMX™86 OPERATOR'S MANUAL now contains this information), adds introductory chapters, describes wild-card support, corrects technical and typographical errors, and documents Release 5 of the iRMX™86 Operating System.	11/82

Additional copies of this manual or other Intel literature may be obtained from:

Literature Department  
Intel Corporation  
3065 Bowers Avenue  
Santa Clara, CA 95051

The information in this document is subject to change without notice.

Intel Corporation makes 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 nor 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 defined in ASPR 7-104.9(a) (9).

No part of this document may be copied or reproduced in any form or by any means without prior written consent of Intel Corporation.

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

BXP  
CREDIT  
i  
ICE  
iCS  
I<sup>2</sup>-ICE  
im  
iMMX

Insite  
Intel  
Intel  
Intelevison  
intelligent Identifier  
intelligent Programming  
Intellec  
Intellink  
iOSP  
iPDS

iRMX  
iSBC  
iSBX  
iSXM  
Library Manager  
MCS  
Megachassis  
Micromainframe  
Micromap

Multibus  
Multichannel  
Multimodule  
Plug-A-Bubble  
PROMPT  
RMX/80  
RUPI  
System 2000  
UPI

## PREFACE

This manual documents the Human Interface, one of the layers of the iRMX 86 Operating System. It is intended for programmers who wish to write application programs that can be loaded and executed via keyboard commands. This manual is divided into the following chapters:

Chapters 1 and 2	Overview of the Human Interface and discussion of the multi-access support.
Chapters 3 through 6	Discussions of the four general categories of Human Interface system calls and how to use them when writing commands.
Chapter 7	Description of the necessary elements of a Human Interface command, as well as the required compilation and link sequences.
Chapter 8	Detailed description of the Human Interface System calls.
Chapter 9	Description of the configurable options of the Human Interface.
Appendixes A through C	Listings of type definitions, exception codes, and string table format.

This manual does not describe the commands supplied with the Human Interface. For information about those commands, refer to the iRMX 86 OPERATOR'S MANUAL (see "Related Publications" section for publication number).

### CONVENTIONS

This manual is intended for the person who designs and implements the commands (the programmer), not for the person who invokes the commands at the terminal. Whenever this manual describes how certain Human Interface features affect the person that invokes the commands, it refers to that person explicitly as the operator.

This manual uses the following notational conventions to illustrate syntax:

UPPERCASE	In examples of system call syntax, uppercase information must be entered exactly as shown. The programmer can, however, enter this information in uppercase or lowercase.
-----------	---

## PREFACE (continued)

- lowercase      In examples of system call syntax, lowercase fields indicate information to be supplied by the programmer. The programmer must enter the appropriate value or symbol for lowercase fields.
- <>              Angle brackets surround variable fields in messages displayed by the Operating System. This information can vary from message to message.

All numbers, unless otherwise noted, are assumed to be decimal. Hexadecimal numbers include the "h" radix character (for example OFFh).

### RELATED PUBLICATIONS

The following manuals provide additional background and reference information.

- Introduction to the iRMX™ 86 Operating System, Order Number: 9803124
- iRMX™ 86 Operator's Manual, Order Number: 144523
- iRMX™ 86 Nucleus Reference Manual, Order Number: 9803122
- iRMX™ 86 Basic I/O System Reference Manual, Order Number: 9803123
- iRMX™ 86 Extended I/O System Reference Manual, Order Number: 143308
- iRMX™ 86 Loader Reference Manual, Order Number: 143318
- iRMX™ 86 Configuration Guide, Order Number: 9803126
- iRMX™ 86 Programming Techniques, Order Number: 142982
- User's Guide for the iSBC® 957B iAPX 86,88 Interface And Execution Package, Order Number: 143979
- iAPX 86,88 Family Utilities Users' Guide, Order Number: 121616
- iMMX™ 800 Multibus® Message Exchange Reference Manual, Order Number: 144912

## CONTENTS

	PAGE
CHAPTER 1	
OVERVIEW	
Supplied Commands.....	1-2
Human Interface System Calls.....	1-2
Standard Initial Program.....	1-3
Multi-Access Support.....	1-3
Wild-Card Pathnames.....	1-4
 CHAPTER 2	
SUPPORTING MULTIPLE TERMINALS	
Communicating with Terminals via the Basic and Extended I/O Systems	2-1
Using the Multi-Access Human Interface.....	2-1
Standard Initial Program.....	2-2
Customized Initial Program.....	2-3
 CHAPTER 3	
COMMAND PARSING	
Standard Command-Line Structure.....	3-1
Parsing the Command Line.....	3-5
Parsing Input and Output Pathnames.....	3-5
Wild-Card Characters in Input and Output Pathnames.....	3-8
Parsing Other Parameters.....	3-10
Parsing Nonstandard Command Lines.....	3-13
Variations on the Standard Command Line.....	3-13
Other Nonstandard Command Lines.....	3-15
Changing the Parsing Buffer.....	3-15
Obtaining the Command Name.....	3-17
 CHAPTER 4	
I/O AND MESSAGE PROCESSING	
Establishing Input and Output Connections.....	4-1
Using C\$GET\$INPUT\$CONNECTION.....	4-1
Using C\$GET\$OUTPUT\$CONNECTION.....	4-1
Example Program Scenario.....	4-2
Communicating with the Operator's Terminal.....	4-3
Formatting Exception Codes into Messages.....	4-4
 CHAPTER 5	
COMMAND PROCESSING	
Creating a Command Connection.....	5-1
Sending Command Lines to the Command Connection and Invoking the Command.....	5-2
Deleting the Command Connection.....	5-3
Example.....	5-3

CONTENTS (continued)

	PAGE
CHAPTER 6	
PROGRAM CONTROL	
How the Default Control-C Mechanism Works.....	6-1
Providing Your Own Control-C Mechanism.....	6-1
CHAPTER 7	
CREATING HUMAN INTERFACE COMMANDS	
Elements of a Human Interface Command.....	7-1
Parsing the Command Line.....	7-1
Avoiding the Use of Certain System Calls.....	7-2
Terminating the Command.....	7-2
INCLUDE Files.....	7-2
Producing an Executable Command.....	7-3
CHAPTER 8	
HUMAN INTERFACE SYSTEM CALLS	
C\$CREATE\$COMMAND\$CONNECTION.....	8-4
C\$DELETE\$COMMAND\$CONNECTION.....	8-8
C\$FORMAT\$EXCEPTION.....	8-9
C\$GET\$CHAR.....	8-11
C\$GET\$COMMAND\$NAME.....	8-13
C\$GET\$INPUT\$CONNECTION.....	8-15
C\$GET\$INPUT\$PATHNAME.....	8-20
C\$GET\$OUTPUT\$CONNECTION.....	8-25
C\$GET\$OUTPUT\$PATHNAME.....	8-31
C\$GET\$PARAMETER.....	8-34
C\$SEND\$COMMAND.....	8-38
C\$SEND\$CO\$RESPONSE.....	8-45
C\$SEND\$EO\$RESPONSE.....	8-48
C\$SET\$PARSE\$BUFFER.....	8-51
CHAPTER 9	
CONFIGURATION OF THE HUMAN INTERFACE	
Resident Configuration.....	9-1
Nonresident Configuration.....	9-2
APPENDIX A	
HUMAN INTERFACE TYPE DEFINITIONS.....	A-1
APPENDIX B	
HUMAN INTERFACE EXCEPTION CODES.....	B-1



CONTENTS (continued)

	PAGE
APPENDIX C	
STRING TABLE FORMAT.....	C-1

TABLES

8-1. System Call Dictionary.....	8-2
A-1. Type Definitions.....	A-1
B-1. Human Interface Exception Codes.....	B-1
B-2. Exception Code Ranges.....	B-2
B-3. iRMX™ 86 Condition Codes.....	B-3

FIGURES

3-1. C\$GET\$INPUT\$PATHNAME and C\$GET\$OUTPUT\$PATHNAME Example.....	3-7
3-2. C\$GET\$PARAMETER Example.....	3-12
5-1. Command Connection Example.....	5-3
C-1. String Table Format.....	C-1



## CHAPTER 1. OVERVIEW

The iRMX 86 Human Interface is a layer of the Operating System that allows console operators to load and execute program files (also called commands) from terminals. When the Human Interface begins running, it:

- Creates an iRMX 86 job for each terminal configured in the Human Interface. This job (also called the interactive job) is the application environment; all commands entered by the operator run as offspring jobs of the operator's interactive job.
- Assigns an area of main memory for the operator (this occurs as part of creating the interactive job). Any commands that the operator runs use this area of memory.
- Starts an initial program (this also occurs as part of creating the interactive job). The initial program is the operator's interface to the Operating System. It is a command line interpreter (CLI), a program that reads its instructions from the terminal. The Human Interface supplies a standard initial program which reads commands from the terminal and invokes the commands based on that terminal input. You can also supply your own initial programs. In fact, there can be a separate initial program for each terminal, if necessary.

When an operator enters information at a Human Interface terminal, the operator communicates with the initial program. With the standard initial program, the operator invokes a command by specifying the pathname of the file that contains the command (and optionally specifying parameters). The initial program reads the information from the terminal and invokes Human Interface system calls to load the command into main memory from secondary storage, create an iRMX 86 job for the command (as an offspring of the operator's interactive job), and begin command execution.

The Human Interface provides several features that aid both operators and programmers. These features include:

- A set of Intel-supplied commands.
- A group of system calls to aid programmers in writing their own commands.
- A standard command line interpreter (CLI).
- Multi-access support.
- Support for wild-card pathnames.

This chapter provides an overview of these features.

## OVERVIEW

### SUPPLIED COMMANDS

In addition to the code for the resident Human Interface, Intel supplies a variety of commands which you can use on any application system that includes the Human Interface. Included are:

- File management commands (such as COPY, DELETE, BACKUP, RESTORE, and others)
- Device and volume management commands (such as ATTACHDEVICE, FORMAT, DISKVERIFY, and others)
- General Utility commands (such as DEBUG, DATE, SUBMIT, and others)

The iRMX 86 OPERATOR'S MANUAL contains complete descriptions of all commands supplied with the Human Interface.

### HUMAN INTERFACE SYSTEM CALLS

The Human Interface provides a set of system calls that programmers can use in commands they write. The following categories of system calls are available:

- Command-parsing system calls
- I/O and message-processing system calls
- Command-processing system calls
- Program control system calls

The command parsing system calls provide the ability to parse the command line, allowing you to determine the parameters that the operator entered when invoking the command. They also allow you to determine the command name and parse other buffers of text. Chapter 3 provides further discussion of the command parsing system calls.

The I/O and message processing system calls allow you to establish connections to input and output files, communicate with the terminal, and format exception codes into a ready-to-display form. Chapter 4 provides a further discussion of the I/O and message processing system calls.

The command processing system calls allow you to invoke interactive commands programmatically. Chapter 5 provides a further discussion of the command processing system calls.

The program control system call allows you to override the default Control-C handling task provided by the Human Interface. Chapter 6 provides a further discussion of program control.

## OVERVIEW

### STANDARD INITIAL PROGRAM

As stated previously, when an operator activates a terminal, the Human Interface assigns an initial program to the operator. This initial program is the first program to run. The identity of this initial program is determined by a privileged operator (normally called the system manager) when adding new users to the system. This process is described in the iRMX 86 CONFIGURATION GUIDE.

Although the initial program can be almost anything -- from an editor to a Basic interpreter -- the Human Interface supplies a standard initial program called the Human Interface command line interpreter (CLI). The function of the Human Interface CLI is to read input from the terminal and invoke commands based on that input. This CLI (or a user-supplied CLI) is required to allow an operator to invoke commands.

### MULTI-ACCESS SUPPORT

The Basic I/O System supports multiple terminals by providing device drivers that communicate with multiple-terminal hardware. The Human Interface adds to this support by providing identification and protection of users based on user IDs. This support is called multi-access support.

With multi-access support, multiple operators can communicate with the Operating System. The Human Interface assigns each operator a unique identification, called a user ID, and a separate area of memory in which to run commands. When an operator creates files or attaches devices, the Human Interface marks the user as the owner of those files or devices. Access to the files by other users depends on the permission granted those users by the owner.

In order to run a multi-access Human Interface, a privileged operator (the system manager) must first set up the proper directory structure and provide several files containing information about the operators that can access the system. This process is described in the iRMX 86 CONFIGURATION GUIDE.

Programmers who write Human Interface commands do not have to write their code differently for a multi-access Human Interface than for a single-access Human Interface. The only difficulty a command might experience in a multi-access environment that it wouldn't experience in a single-access environment involves accessing files and devices. When a command is invoked by an operator, the command inherits the operator's user ID. Thus the command can perform operations only on files and devices to which the invoking operator has access.

WILD-CARD PATHNAMES

The Human Interface supports the use of wild-card characters in file names when the file names are parameters of most commands. This gives the operator a shorthand method of specifying several files in a single reference. The wild-card characters supported by the Human Interface are:

- ? Matches any single character
- \* Matches any sequence of characters (including zero characters)

The iRMX 86 OPERATOR'S MANUAL describes how an operator can use wild-card characters when entering commands.

Programmers who write their own Human Interface commands do not have to provide special code to support wild-card pathnames as long as they use the Human Interface system calls C\$GET\$INPUT\$PATHNAME and C\$GET\$OUTPUT\$PATHNAME to obtain the file names from the command line. The Human Interface contains the mechanism to interpret the wild cards and return the correct file name to the command. Refer to Chapter 3 for more information about these system calls.

\*\*\*

## CHAPTER 2. SUPPORTING MULTIPLE TERMINALS

The iRMX 86 Operating System provides two ways for you to implement multiple-terminal support on your application system. You can:

- Write application tasks that use the system calls of the Basic and Extended I/O Systems to communicate directly with multiple terminals.
- Use the multi-access Human Interface.

This chapter discusses both methods.

### COMMUNICATING WITH TERMINALS VIA THE BASIC AND EXTENDED I/O SYSTEMS

One method of providing multiple terminal support is to omit the Human Interface from your system, write your own application programs that access the terminals directly, and configure these programs as tasks in the Operating System. The Basic I/O System provides device drivers that allow tasks to communicate with multiple terminals. Therefore, if your system contains the necessary hardware, your application tasks can use Basic and Extended I/O System calls to communicate with each terminal in your system.

If you communicate with the terminals directly, without using the Human Interface, you can tailor your terminal interface to meet your exact needs. This might result in smaller, faster code than the Human Interface (but at the expense of an increased program development effort). This method requires you to write a great deal of code that the Human Interface already supplies.

If you plan to use this method of providing multiple terminal support, none of the information contained in this manual applies to you. Refer to the iRMX 86 BASIC I/O SYSTEM REFERENCE MANUAL and the iRMX 86 EXTENDED I/O SYSTEM REFERENCE MANUAL for information about the system calls you can use to communicate with terminals.

### USING THE MULTI-ACCESS HUMAN INTERFACE

The other method of providing multiple-terminal support is to use the multi-access support provided by the Human Interface. The multi-access support includes code required to communicate with multiple terminals. It uses the same Basic and Extended I/O System calls that you would have to use if you implemented the method described in the previous section. However, the multi-access Human Interface also provides high-level support for this communication. For example, from a terminal in a multi-access system, an operator can execute commands, run development programs (like editors, compilers, and so on), and run other application programs.

If you decide to use the multi-access support features of the Human Interface, you can still tailor your system to meet your individual needs. An important way of doing this is by selecting, for each operator, the initial program that runs when that operator accesses the Human Interface. There are two choices: the initial program supplied with the Human Interface (the standard CLI) or initial programs that you write. The user description files maintained by the system manager identify this choice to the Human Interface (refer to the iRMX 86 CONFIGURATION GUIDE for more information). By selecting the initial program, you can greatly influence the operator's interface to the Human Interface.

#### STANDARD INITIAL PROGRAM

The Human Interface supplies a command line interpreter (CLI) as the standard initial program. During initialization, the Human Interface CLI performs the following operations:

- Displays a sign-on message.
- Creates an iRMX 86 object called a command connection in which it places information received from the terminal. Refer to Chapter 5 for more information about command connections.
- Attaches or creates the operator's :PROG: directory.
- Submits the file :PROG:R?LOGON for processing.

After this initial processing, the Human Interface CLI performs the following operations:

- Displays the Human Interface prompt (-) and reads input from the terminal (using the Human Interface system call C\$SEND\$CO\$RESPONSE).
- Places the information it reads into the command connection (using the Human Interface system call C\$SEND\$COMMAND). After receiving a complete command, the command connection removes the command name portion, loads the file containing the command, and passes the parameters to the command.
- Recognizes continuation lines and displays a different prompt (\*\*) when a continuation line is required.
- Displays error messages in the event of certain operator errors.

This is the environment described in the iRMX 86 OPERATOR'S MANUAL. If it satisfies the needs of your application system, you can assign the Human Interface CLI to each operator as an initial program.



## SUPPORTING MULTIPLE TERMINALS

### CUSTOMIZED INITIAL PROGRAMS

If the standard initial program does not meet your needs, you have the option of providing your own initial programs. These initial programs might be similar to the Human Interface CLI, or they might be completely different kinds of programs. For example, you could write a CLI that allows access to files in selected directories only. This would prevent an operator from accidentally modifying other files. Or if you want a particular operator to use only Basic-language programs, a Basic interpreter might be the initial program for that operator. You can select the initial program for each operator. You specify this selection in the user description files maintained by the system manager (refer to the iRMX 86 CONFIGURATION GUIDE).

If you provide your own initial program, this program must obey the following rules:

- It must perform input and output via logical names :CI: and :CO:.
- If it requires the ability to run Human Interface commands, it must create an iRMX 86 object called a command connection (via the C\$CREATE\$COMMAND\$CONNECTION system call). If the initial program does not create a command connection, it (and any other application tasks) cannot use the following Human Interface system calls:

```
C$GET$INPUT$PATHNAME
C$GET$OUTPUT$PATHNAME
C$SEND$CO$RESPONSE
C$SEND$EO$RESPONSE
C$SEND$COMMAND
C$DELETE$COMMAND$CONNECTION
```

- If it does not create a command connection but still wishes to use the Human Interface system calls C\$GET\$PARAMETER and C\$GET\$CHAR, it must first invoke the C\$SET\$PARSE\$BUFFER system call.
- If it receives an end-of-file indication from the terminal, it must terminate processing.
- It must invoke the Extended I/O System call EXIT\$IO\$JOB to terminate processing. It must not use the PL/M-86 or ASM86 RETURN statement for this purpose.

Refer to Chapter 8 for detailed descriptions of the Human Interface system calls mentioned in this section. Refer to the iRMX 86 EXTENDED I/O SYSTEM REFERENCE MANUAL for information about the EXIT\$IO\$JOB system call.

\*\*\*



## CHAPTER 3. COMMAND PARSING

Whenever a Human Interface operator enters characters at a terminal to invoke a command, an initial program associated with that operator reads that information and causes the Operating System to invoke the command. When it invokes the command, the Operating System places the parameters into a parsing buffer. One of the first things that the command must do is to read the parsing buffer, break the command line into individual parameters, and determine the correct action to take based on the number and meaning of the parameters.

The Human Interface provides several system calls to parse command lines that follow a standard structure. It also provides other system calls to process nonstandard formats. This chapter:

- Lists the standard structure of command lines
- Describes the system calls used to parse commands of this structure
- Discusses how to switch the parsing buffer
- Describes the system calls that you can use to parse nonstandard commands
- Describes a system call that you can use to obtain the command name the operator used when invoking the command

### STANDARD COMMAND-LINE STRUCTURE

The standard structure of a Human Interface command line consists of a number of elements separated by spaces. It is recommended that your commands follow this structure. However, if you require a different structure, refer to the "Parsing Nonstandard Command Lines" section of this chapter. The standard structure is as follows (square brackets [] indicate optional portions):

```
command-name [inpath-list [preposition outpath-list]] [parameters] cr
```

where:

command-name Pathname of the file containing the command's executable object code.

**inpath-list** One or more pathnames, separated by commas, of files that the Human Interface reads as input during command execution. Individual pathnames can contain wild-card characters to signify multiple files. Refer to the iRMX 86 OPERATOR'S MANUAL for a description of the wild-card characters and their usage. You can use the C\$GET\$INPUT\$PATHNAME system call to process this inpath-list.

**preposition** A word that tells the Human Interface how to handle the output. The standard structure supports the following prepositions:

**TO** The Human Interface writes the output to a new file indicated by the output pathname. If the file already exists, the Human Interface queries the operator as follows:

<pathname>, already exists, OVERWRITE?

If the operator enters a Y or an R (uppercase or lowercase), the Human Interface replaces the existing file with the new output. Any other character causes the Human Interface to proceed with the next pair of input and output files.

**OVER** The Human Interface writes the output to the file indicated by the output pathname. It overwrites any information that currently exists in the file.

**AFTER** The Human Interface appends the output to the end of the file indicated by the output pathname.

You can use the C\$GET\$OUTPUT\$PATHNAME system call to process the preposition.

**outpath-list** One or more pathnames, separated by commas, of files that are to receive the output during command execution. The total number of pathnames in this list and the number of wild cards used depends on the inpath-list. Refer to the iRMX 86 OPERATOR'S MANUAL for more information. You can use the C\$GET\$OUTPUT\$PATHNAME system call to process the outpath-list.

## COMMAND PARSING

**parameters** Parameters that cause the command to perform additional or extended services during command execution. The standard structure supports parameters with the following formats:

- value-list** The parameter consists solely of one or more groups of characters (called values) separated by commas. When the value-list is present in the command line, the command performs the service requested by the values.
- keyword=value-list** A keyword with an associated value (or list of values, separated by commas). The keyword portion identifies the kind of service to perform, and each value supplies further information about the service.
- keyword(value-list)** Alternate form of the previous format.
- keyword value-list** A keyword with an associated value (or list of values, separated by commas). Like the previous two formats, the keyword portion identifies the kind of service to perform and each value portion provides more information about the service. However, the keyword must be identified to the command as a preposition (refer to the description of the C\$GET\$PARAMETER system call for more information).

You use the C\$GET\$PARAMETER system call to process the outpath-list.

**cr** Line terminator character. The RETURN (or CARRIAGE RETURN) key and NEW LINE (or LINE FEED) key are both line terminators.

The Human Interface also supports the following special characters:

**continuation character** An ampersand character (&). When an operator includes an ampersand in the command line as the last character before the line terminator, the Human Interface assumes that the command invocation continues on the next line. If the standard Human Interface command line interpreter (or any custom command line interpreter that uses C\$SEND\$COMMAND to invoke commands) processes the operator's command entry, the ampersand (and the line terminator that follows) are

edited out of the parsing buffer. Then the continuation line is read and appended to the parsing buffer. This process continues until the operator enters a line without a continuation character. Therefore, when the command receives control, its parsing buffer contains a single command invocation, without intermediate continuation characters or line terminators.

comment  
character

A semicolon character (;). The Human Interface considers this character and all text that follows it on a line to be a non-executable comment. If the standard Human Interface command line interpreter (or any custom command line interpreter that uses C\$SEND\$COMMAND to invoke commands) processes the operator's command entry, all comments are edited out of the parsing buffer. Therefore, individual commands do not have to search for and discard comments.

quoting  
characters

Two single-quote (') or double-quote (") characters remove the semantics of special characters they surround (but you must use the same character for both the beginning and ending quote). If a command line contains quoted characters, the Human Interface system calls that invoke the command and parse the command line do not perform any special functions associated with the surrounded characters. For example, an ampersand surrounded by double quotes is interpreted as a single ampersand and not a continuation character.

The quotes remove the semantics of characters that are special to the Human Interface but not special to other layers of the Operating System. Therefore quotes do not remove the semantics of characters such as :, /, and ^, which are special to the I/O System.

To include the quoting character in the quoted string, the operator must specify the character twice or use the other quoting character. For example:

'can't' or "can't"

causes:

can't

to be entered in the command line.

## COMMAND PARSING

### PARSING THE COMMAND LINE

When a command begins executing, a parsing buffer associated with the command contains all the parameters that the operator entered when invoking the command (everything except the command-name portion of the invocation line). The Human Interface maintains a pointer for this parsing buffer which initially points to the first parameter. By invoking any of the following Human Interface system calls, the command can read the parameters from the parsing buffer:

```
C$GET$INPUT$PATHNAME
C$GET$OUTPUT$PATHNAME
C$GET$PARAMETER
C$GET$CHAR
```

Each of the first three system calls reads an entire parameter and causes the Human Interface to move the pointer to the next parameter. These system calls understand quoting characters, remove the special meaning from quoted characters, and discard the quote characters.

The last system call, C\$GET\$CHAR, sees the parsing buffer as string of characters. It reads a single character and causes the Human Interface to move the pointer to the next character. It does not understand the notion of quoting characters; therefore it does not remove the special meaning from quoted characters, nor does it skip over the quotes. Except for positioning the parsing pointer to a particular place in the buffer, C\$GET\$CHAR should not be used with the first three system calls.

### PARSING INPUT AND OUTPUT PATHNAMES

If you restrict the invocation lines of the commands you write to a form that is similar to the standard format discussed earlier in this chapter, you can use the system calls C\$GET\$INPUT\$PATHNAME and C\$GET\$OUTPUT\$PATHNAME to identify the input and output pathnames in the command line. Since the command line can contain multiple pathnames, you might have to invoke these system calls several times to obtain all the pathnames. However, the first call to C\$GET\$INPUT\$PATHNAME reads the entire inpath-list (the list of pathnames separated by commas) into a buffer, moves the parsing pointer to the next parameter, and returns the first pathname to the command. Likewise, the first call to C\$GET\$OUTPUT\$PATHNAME notes the preposition (TO, OVER, or AFTER), reads the entire outpath-list into a buffer, moves the parsing pointer to the parameter after the outpath-list, and returns the first pathname to the command. Succeeding C\$GET\$INPUT\$PATHNAME and C\$GET\$OUTPUT\$PATHNAME calls return additional pathnames from the buffers created previously, but they do not move the parsing pointer to the next parameter.

For example, if the parsing buffer contains:

```
A,B TO C,D
```

the first call to `C$GET$INPUT$PATHNAME` obtains both input pathnames (A and B) and returns the first one (A) to the caller. The first call to `C$GET$OUTPUT$PATHNAME` obtains both output pathnames (C and D) and returns the first one (C) to the caller. `C$GET$OUTPUT$PATHNAME` also identifies TO as the preposition.

These system calls handle single pathnames, lists of pathnames, and pathnames containing wild-card characters. However, because of this versatility and because output pathnames are dependent on input pathnames when both use wild-card characters, you must make calls to `C$GET$INPUT$PATHNAME` and `C$GET$OUTPUT$PATHNAME` in a particular order. To use these system calls effectively, obey the following rules:

1. Always call `C$GET$INPUT$PATHNAME` to obtain the input pathname before calling `C$GET$OUTPUT$PATHNAME` to obtain the corresponding output pathname. This is necessary because with wild-card characters, the identity of the output pathname depends on the identity of the input pathname. Therefore, `C$GET$OUTPUT$PATHNAME` cannot determine the output pathname until `C$GET$INPUT$PATHNAME` determines the corresponding input pathname.
2. Always interleave your calls to `C$GET$INPUT$PATHNAME` and `C$GET$OUTPUT$PATHNAME`. This is necessary to handle wild-card characters and lists of pathnames. If you invoke two calls to `C$GET$INPUT$PATHNAME` without an intermediate call to `C$GET$OUTPUT$PATHNAME`, you will not be able to obtain the first output pathname. Similarly, if you invoke two calls to `C$GET$OUTPUT$PATHNAME` without an intermediate call to `C$GET$INPUT$PATHNAME`, the second call returns invalid information.

`C$GET$INPUT$PATHNAME` and `C$GET$OUTPUT$PATHNAME` return the pathnames in the form of strings. Each string is a group of bytes in which the first byte contains the number of bytes that follow. For these system calls, the remaining bytes in the string contain the pathname. If `C$GET$INPUT$PATHNAME` returns a zero-length string (that is, the first byte is zero), you know that there are no more pathnames to obtain.

After calling `C$GET$INPUT$PATHNAME` and `C$GET$OUTPUT$PATHNAME` to obtain the input file and corresponding output file, you can use the system calls `C$GET$INPUT$CONNECTION` and `C$GET$OUTPUT$CONNECTION` to obtain connections to those files. Chapter 4 contains more information about `C$GET$INPUT$CONNECTION` and `C$GET$OUTPUT$CONNECTION`. Upon obtaining connections to the files, you can perform the necessary I/O operations.

Figure 3-1 contains an example of a program that uses `C$GET$INPUT$PATHNAME` and `C$GET$OUTPUT$PATHNAME` in its command-line parsing (it also uses `C$GET$INPUT$CONNECTION` and `C$GET$OUTPUT$CONNECTION` to obtain connections to the files. This command is a partial example of a COPY command that you could implement.



COMMAND PARSING

```

/*****
*   This example demonstrates the use of the following Human Interface   *
*   system calls:                                                         *
*                                                                           *
*       rq$C$get$input$pathname                                           *
*       rq$C$get$output$pathname                                          *
*       rq$C$get$input$connection                                         *
*       rq$C$get$output$connection                                        *
*                                                                           *
*   This program is a possible implementation of a COPY utility whose    *
*   purpose is to copy data from successive input files to corresponding *
*   output files.  For example, to copy file A to file B, file C to file *
*   D, and file E to file F, an operator could specify the following     *
*   command line:                                                         *
*                                                                           *
*       COPY A,C,E TO B,D,F                                              *
*****/

```

copy: DO;

```

$include (hexcep.lit)
$include (iexioj.ext)
$include (hgticn.ext)
$include (hgtipn.ext)
$include (hgtocn.ext)
$include (hgtopn.ext)

```

```

DECLARE (input$pathname, output$pathname) structure (
                                length      byte,
                                char (41)   byte ),
    output$prep      byte,
    (input$token, output$token) word,
    excep           word,
    exitexcep      word;

```

```

/* Get the first input pathname string */
CALL rq$C$get$input$pathname (@input$pathname, SIZE(input$pathname),
                              @excep);
IF excep <> E$OK      THEN
    CALL rq$exit$io$job (exitexcep, 0, @excep);

DO WHILE (input$pathname.length <> 0); /* A zero length indicates no more
                                        input parameters.                */

/* Get the corresponding output pathname string */
output$prep = rq$C$get$output$pathname (@output$pathname,
                                         SIZE(output$pathname),
                                         @(7,'TO :CO:'), @excep);

IF excep <> E$OK      THEN
    CALL rq$exit$io$job (exitexcep, 0, @excep);

```

Figure 3-1. C\$GET\$INPUT\$PATHNAME and C\$GET\$OUTPUT\$PATHNAME Example

```
/* Establish connection with the pair of input and output files */
input$token = rq$C$get$input$connection (@input$pathname, @excep);
IF excep <> E$OK THEN
    CALL rq$exit$io$job (exitexcep, 0, @excep);

output$token = rq$C$get$output$connection (@output$pathname,
                                           output$prep, @excep);
IF excep <> E$OK THEN
    CALL rq$exit$io$job (exitexcep, 0, @excep);

    .
    .      Code to copy data and close both files
    .

/* Get the next input pathname string */
CALL rq$C$get$input$pathname (@input$pathname, SIZE(input$pathname),
                              @excep);
IF excep <> E$OK THEN
    CALL rq$exit$io$job (exitexcep, 0, @excep);

END /* DO WHILE */

/* Finish I/O processing */
CALL rq$exit$io$job (exitexcep, 0, @excep);

END copy;
```

Figure 3-1. C\$GET\$INPUT\$PATHNAME and C\$GET\$OUTPUT\$PATHNAME Example  
(continued)

#### WILD-CARD CHARACTERS IN INPUT AND OUTPUT PATHNAMES

Wild-card characters provide a shorthand notation for specifying several files in a single reference. The Human Interface supports two wild-card characters for use in the last component of input or output pathnames. The wild-card characters are:

? The question mark matches any single character. For example, the name "FILE?" could imply all of the following names (and more):

```
FILE1
FILE2
FILEX
```

## COMMAND PARSING

- \* The asterisk matches any sequence of characters (including zero characters). For example, the name "\*FILE" could imply all of the following files (and more):

```
OBJECTFILE
FILE
V1.2FILE
AFILE
```

The IRMX 86 OPERATOR'S MANUAL describes how to use wild-card characters when entering commands. It also discusses restrictions and operational characteristics of which an operator should be aware. Refer to that manual for more information about using wild-card characters in file names.

The C\$GET\$INPUT\$PATHNAME and C\$GET\$OUTPUT\$PATHNAME system calls automatically handle pathnames that contain wild-card characters. They treat a wild-carded pathname as a list of pathnames.

C\$GET\$INPUT\$PATHNAME matches wild cards. That is, each time you call it, it compares the wild-carded component with the files in the specified directory and returns the pathname the next file that matches. For example, if an input pathname is:

```
:PROG:PLM/A*
```

C\$GET\$INPUT\$PATHNAME searches the :PROG:PLM directory and returns the pathname of the next file that begins with the letter "A."

C\$GET\$OUTPUT\$PATHNAME generates wild cards. Each time you call it, it compares the wild-carded output pathname with the wild-carded input pathname and with the most recent pathname returned by C\$GET\$INPUT\$PATHNAME. Then it generates a corresponding output pathname based on that information. The output pathname could refer to an existing file or to a file which does not yet exist.

As an example, suppose an operator's default directory contains the following files:

ALPHA	BETA
All	Bll
ADAM	Cll

Now suppose that you have written a command called REFINE that reads some information from an input file, adjusts that information in some manner, and writes the information to an output file. Assuming that you interleaved the calls to C\$GET\$INPUT\$PATHNAME and C\$GET\$OUTPUT\$PATHNAME correctly when you wrote the command, an operator could enter a command line as follows:

```
REFINE A*,B* TO C*,D*
```

In this case, C\$GET\$INPUT\$PATHNAME and C\$GET\$OUTPUT\$PATHNAME return pathnames as follows:

<u>Pathname list returned by C\$GET\$INPUT\$PATHNAME</u>	<u>Corresponding pathname list returned by C\$GET\$OUTPUT\$PATHNAME</u>
ALPHA	CLPHA
All	Cll
ADAM	CDAM
BETA	DETA
Bll	Dll

### PARSING OTHER PARAMETERS

The C\$GET\$PARAMETER system call is also available for parsing command lines of the standard format. You can use this system call for the following purposes:

- To parse parameters which appear after the input and output pathnames.
- To parse all parameters, if the command does not use input and output files.
- To parse the input and output pathnames, if the command requires a preposition other than TO, OVER, or AFTER.

If you use C\$GET\$PARAMETER to parse input and output pathnames, you must provide additional code to handle wild-card characters that may appear in the command line. This is unlike C\$GET\$INPUT\$PATHNAME and C\$GET\$OUTPUT\$PATHNAME which handle wild-card characters automatically. For example, suppose a command line contains the pathname:

FILE\*

If you use C\$GET\$INPUT\$PATHNAME to parse this parameter, the system call assumes that FILE\* is a wild-carded pathname. It searches the operator's default directory and returns the pathname of the first file whose name starts with the characters "FILE". Subsequent calls to C\$GET\$INPUT\$PATHNAME return other pathnames that meet the conditions.

However, if you use C\$GET\$PARAMETER to parse the same parameter, the system call returns the value:

FILE\*

It does not know that the characters represent a pathname, nor does it know that the asterisk represents a wild card.

When called, C\$GET\$PARAMETER parses a single parameter and moves the pointer of the parsing buffer to the next parameter. The parameter returned as a result of this call can be in any of the following forms:

## COMMAND PARSING

value-list	A value or group of values separated by commas. The system call returns the entire list in the form of a string table (described in Appendix C). It places each of the values in the value list in a separate string.
keyword = value-list or keyword (value-list)	A keyword indicating the kind of parameter, followed by a value (or group of values, separated by commas). The presence of the equal sign or the parentheses lets the system call recognize keyword parameters without foreknowledge of the keywords. It also informs the system call that the characters following the equal sign (or the characters in parenthesis) represent a value-list and not a separate parameter. The system call returns the keyword in a string and the value-list in a string table.
keyword value-list	A keyword indicating the kind of parameter, followed by a value (or group of values, separated by commas). In this case, since the keyword and value-list are separated by spaces instead of by an equal sign or parentheses, the keyword is referred to as a preposition. In order for the system call to recognize that this structure is a keyword/value-list instead of two separate parameters, you must supply, as input to the system call, a string table containing all the possible prepositions that could occur. The system call checks this list to determine whether a group of characters separated by spaces is a preposition keyword or a separate parameter.

Individual parameters are separated by spaces.

In general, the value-list of a parameter is either a single value or a list of values separated by commas. C\$GET\$PARAMETER returns each of these values as a string in a string table. However, an individual value can itself consist of a value-list. If a group of values (separated by commas) is enclosed in parentheses, C\$GET\$PARAMETER treats the values as a single value, returning them in single string. For example, in the following value-list:

A,(B,C,D),E

C\$GET\$PARAMETER considers "B,C,D" as a single value. Therefore, the value-list consists of three values: "A", "B,C,D", and "E".

Figure 3-2 contains an example of a program that uses C\$GET\$PARAMETER in its command-line parsing.

```

/*****
*   This example demonstrates the use of the following Human Interface   *
*   system call:                                                         *
*                                                                           *
*       rq$C$get$parameter                                               *
*                                                                           *
*   This program makes use of rq$C$get$parameter to parse a keyword     *
*   parameter in a command line. Here, the keyword, "SIZE", is parsed   *
*   and its value portion converted to a word value and placed in       *
*   "size$val". For example, an operator could specify the following    *
*   command line:                                                         *
*                                                                           *
*       PROG1  SIZE = 400                                                *
*                                                                           *
*   Note that if the "SIZE" parameter is not present, "size$val"receives *
*   a default value.                                                       *
*****/

progl: D0;

$include (hexcep.lit)
$include (hgtpar.ext)

DECLARE STRING LITERALLY 'STRUCTURE (len BYTE, str (1) BYTE)',
          STRING$TABLE LITERALLY 'STRUCTURE (num$entries BYTE,
                                     entries (1) BYTE)',
          PARAMETER$KEYWORD$MAX LITERALLY '20',
          VALUE$TABLE$MAX LITERALLY '80',
          DEFAULT$SIZE LITERALLY '100';

DECLARE value$table$buf (VALUE$TABLE$MAX) BYTE, /* Receives string table
                                                value */
value$table STRING$TABLE AT (@value$table$buf),
value$str$ptr POINTER,
value$str BASED value$str$ptr STRING; /* For referencing strings
                                        in the string table */

DECLARE parameter$keyword$buf (PARAMETER$KEYWORD$MAX) BYTE, /* Receives
                                                            the keyword
                                                            string */
parameter$keyword STRING AT (@parameter$keyword$buf),
except WORD,
(size$val, i) WORD;

```

Figure 3-2. C\$GET\$PARAMETER Example

## COMMAND PARSING

---

```
/* Get the next parameter, if present */
IF (rq$C$get$parameter (@parameter$keyword, PARAMETER$KEYWORD$MAX,
                        @value$table, VALUE$TABLE$MAX,
                        0,0,
                        @excep) ) THEN
  IF (parameter$keyword.str(0) = 'S') AND /* Is the keyword 'SIZE'? */
      (parameter$keyword.str(1) = 'I') THEN
    DO;
      value$str$ptr = @value$table.entries; /* Point to 1st entry in
                                             table */

      size$val = 0;
      DO i = 0 to value$str.len - 1; /* Convert number string to word
                                     value */
        size$val = size$val * 10;
        size$val = size$val + (value$str.str(i) - 30H);
      END;
    END;
  ELSE
    size$val = DEFAULT$SIZE; /*If the 'SIZE' parameter is not present,
                              use the default size. */

    .
    .   Continue with the rest of the program
    .
```

Figure 3-2. C\$GET\$PARAMETER Example (continued)

---

### PARSING NONSTANDARD COMMAND LINES

If the command lines of the commands you write follow the recommended structure described earlier in this chapter, you can use C\$GET\$INPUT\$PATHNAME, C\$GET\$OUTPUT\$PATHNAME, and C\$GET\$PARAMETER to parse the command line. However, if you require the invocation line to be of a different form, you might not be able to use these system calls. The following sections discuss two types of nonstandard command lines: one that is similar to the standard and one that is completely different.

#### VARIATIONS ON THE STANDARD COMMAND LINE

The "Standard Command-Line Structure" section of this chapter recommends that the first parameters of your commands be a list of input pathnames, a preposition, and a list of output pathnames. This convention allows your commands to know when to use C\$GET\$INPUT\$PATHNAME and C\$GET\$OUTPUT\$PATHNAME system calls and when to use C\$GET\$PARAMETER system calls. With this convention, commands always call C\$GET\$INPUT\$PATHNAME

## COMMAND PARSING

and `C$GET$OUTPUT$PATHNAME` first, before obtaining any optional parameters. Therefore, the input and output pathnames are the only position-dependent parameters in your commands; other parameters can appear in any order and can be optional.

However, suppose you want to structure your commands so that other parameters appear before the input and output pathnames. You can still use `C$GET$INPUT$PATHNAME` and `C$GET$OUTPUT$PATHNAME` to parse the input and output pathnames. But, you have to ensure that your command knows which of the parameters contain the input and output pathnames. You can do this in several ways. Two of them are:

- Enforce a rigid structure on the command line. For example, suppose you want two parameters to appear before the input and output pathnames, such as:

```
command p1 p2 input-pathname prep output-pathname
```

Your command could use `C$GET$PARAMETER` to parse the first and second parameters. Then it could use `C$GET$INPUT$PATHNAME` and `C$GET$OUTPUT$PATHNAME` to parse the input and output pathnames. If you do this, `p1` and `p2` are position-dependent parameters which must be included whenever the command is invoked.

- Use a separate parameter as a switch to inform the command that the parameters that follow are input and output pathnames. This method requires more code to implement but it can allow you to make all your parameters (including the input and output pathnames) position-independent.

For example, you could implement your command such that whenever the operator entered a parameter called `FROM`, it would signal the command that the next parameters were input and output pathnames. This command could contain a main loop that used `C$GET$PARAMETER` to parse parameters. Then, whenever it received a parameter whose value was `"FROM"`, it could call another portion of code that used `C$GET$INPUT$PATHNAME` and `C$GET$OUTPUT$PATHNAME`. After retrieving the input and output pathnames, the code could return to the main loop to continue processing parameters.

A hypothetical command of this sort might be called `RETRIEVE`, a command that retrieves information from various data bases. The operator could invoke this command with a command line such as:

```
RETRIEVE NAMES ADDRESSES PHONES FROM file1 TO file2
```

In this command, operators can specify what they want to retrieve before they specify where to get the information.



## COMMAND PARSING

### OTHER NONSTANDARD COMMAND LINES

In some instances, you might want your command line to look completely different from that described earlier in this chapter. For example, suppose you require a syntax in which the following rules apply:

- Spaces have no significance and can be omitted between parameters.
- You must place a prefix character before each parameter (a \$ indicates an input file, an @ indicates an output file, and a - indicates all other parameters).

With this kind of syntax, a user could invoke a command (in this example the command is again called REFINE) as follows:

```
REFINE $infile-medium@outfile
```

Where infile is the file from which to read information, outfile is the file in which REFINE should place its output, and medium is a parameter that further directs the processing.

If you require the syntax outlined in this example (or any other nonstandard syntax), you cannot use C\$GET\$INPUT\$PATHNAME, C\$GET\$OUTPUT\$PATHNAME, and C\$GET\$PARAMETER to parse the individual parameters. Any of these system calls would return the entire parameter list as a single parameter.

For cases such as this, you can use the C\$GET\$CHAR system call to parse the command line. This system call performs a single, simple operation. It returns a single character from the command line and moves the pointer to the next character. It does not understand the notion of parameters as explained earlier in this chapter. Nor does it understand wild-card characters or quoting characters.

C\$GET\$CHAR requires you to provide the parsing algorithm in your own program, because it makes no assumptions about the structure or order of parameters. However, by using C\$GET\$CHAR you can enforce any command syntax you choose.

Because C\$GET\$CHAR moves the pointer character by character, not parameter by parameter, you should take care when using C\$GET\$CHAR in the same program with C\$GET\$INPUT\$PATHNAME, C\$GET\$OUTPUT\$PATHNAME, and C\$GET\$PARAMETER. You must ensure that C\$GET\$CHAR leaves the pointer pointing at the beginning of a parameter (or at blank characters which immediately precede the parameter) before invoking any of the other system calls.

### CHANGING THE PARSING BUFFER

When a command begins execution, it has a parsing buffer that is set up by the Human Interface to contain the parameters of the command. The command parsing system calls listed in this chapter operate on that parsing buffer. This allows the command to parse its parameters.

## COMMAND PARSING

Some commands might require the ability to parse additional lines of text (for example, an editor needs to parse individual editor commands). A command such as this cannot use the Human Interface-provided parsing buffer because it has no way of placing information in the buffer, and because it cannot reset the parsing pointer to the beginning of the buffer.

To meet the needs of commands such as this, the Human Interface provides a system call to change the parsing buffer from the one the Human Interface provides to one that the command provides. This system call, `C$SET$PARSE$BUFFER`, switches the parsing buffer and sets the parsing pointer to the beginning of the buffer.

One of the parameters of the `C$SET$PARSE$BUFFER` system call (`buff$p`) is a pointer to a buffer containing the text to be parsed. This buffer can contain text read from the terminal, text read from a file, or even text that you "hard code" into the command. After the call to `C$SET$PARSE$BUFFER`, the following command parsing system calls obtain information from the new parsing buffer:

`C$GET$PARAMETER`  
`C$GET$CHAR`

The other command parsing calls (`C$GET$INPUT$PATHNAME` and `C$GET$OUTPUT$PATHNAME`) are not affected by calls to `C$SET$PARSE$BUFFER`. These calls always obtain pathnames from the original parsing buffer (the command line).

When you establish a new parsing buffer, `C$SET$PARSE$BUFFER` sets the parsing pointer to the beginning of the buffer. This allows you to use one buffer for parsing many lines of text. For example, suppose your command has several sub-commands. Each time the operator enters a sub-command, your command reads the sub-command into a buffer, calls `C$SET$PARSE$BUFFER` to reset the parsing pointer, and parses the sub-command. The program flow for an operation like this could be:

1. Read the information from the terminal into a buffer (use `C$SEND$CO$RESPONSE`, `C$SEND$EO$RESPONSE`, or an Extended I/O System call).
2. Call `C$SET$PARSE$BUFFER` to set the parsing buffer to the buffer containing the sub-command. This sets the parsing pointer to the beginning of the buffer.
3. Parse the sub-command using `C$GET$PARAMETER` or `C$GET$CHAR` system calls.
4. Perform the operations requested by the sub-command.
5. Go back to step 1. Continue this loop until the operator exits from the command.

## COMMAND PARSING

If you specify a zero value for the `buff$` parameter of `C$SET$PARSE$BUFFER`, the parsing buffer switches back to the original command line buffer. However, the parsing pointer does not reset to the beginning of the buffer; it remains pointing at the next parameter in the command line. This allows you, if you wish, to parse part of the command line, switch buffers and parse a portion of another buffer, and switch back to the command line.

There is one problem with switching back and forth between parsing buffers. Except when you switch to the command line buffer, every time you call `C$SET$PARSE$BUFFER`, the parsing pointer moves to the start of the buffer. Therefore, you lose your place in the buffer. However, `C$SET$PARSE$BUFFER` returns, in its offset parameter, a value that indicates the position of the pointer in the previous buffer. This value specifies the offset of the pointer, in bytes, from the beginning of the buffer. If you intend to switch back to that buffer (by again calling `C$SET$PARSE$BUFFER`), you can use this value to move the pointer to its previous position.

One way to do this is to use the `C$GET$CHAR` system call to move the parsing pointer back to its previous position. After switching back to the original buffer, call `C$GET$CHAR` the number of times specified in the offset parameter of the first `C$SET$PARSE$BUFFER` call (not the one that switched back to the buffer). This positions the pointer to its previous location. You can then continue parsing parameters from the point at which you left off.

Another way to do this is by treating your parsing buffer as an array of characters (an array called `CHAR`, for example). When you call `C$SET$PARSE$BUFFER` the first time, you can specify the `buff$` parameter to point to the first element of the array (`CHAR(0)`, for example). Then, when you switch parsing buffers, `C$SET$PARSE$BUFFER` returns, in the offset parameter, the number of bytes already parsed. When you switch back to the first parsing buffer, you can use this offset value as an index into the array; that is, have the `buff$` parameter point to `CHAR(offset)`.

### OBTAINING THE COMMAND NAME

A user invokes a command by specifying the pathname of the file containing its object code and any parameters the command requires. The Human Interface places the parameters in a parsing buffer, which the command can access by invoking the system calls described earlier in this chapter. In addition, the Human Interface places the command name in another buffer. The command can obtain this name by calling `C$GET$COMMAND$NAME`.

`C$GET$COMMAND$NAME` does not operate on the parsing buffer used by the other command parsing system calls. Nor is it affected by the `C$SET$PARSE$BUFFER` system. It can be called multiple times; each time it returns the same command name.

## COMMAND PARSING

If the operator enters the complete pathname of the command (including the logical name), the command-name buffer contains exactly what the operator entered. However, if the operator enters a command name without a logical name, the Human Interface automatically searches a number of directories for the command. In this case, the command-name buffer contains not only the name the operator entered, but also the directory containing the command (such as :SYSTEM:, :PROG:, or :\$:).

Therefore, a command can use the value returned by C\$GET\$COMMAND\$NAME and the ampersand pathname separator (^) to access the directory in which it resides. For example, if "command-name" is the name received from C\$GET\$COMMAND\$NAME, a command could access its directory by using the pathname:

```
command-name^
```

It could access another file in the directory by specifying the pathname:

```
command-name^file
```

\*\*\*

## CHAPTER 4. I/O AND MESSAGE PROCESSING

The Human Interface provides several system calls that establish connections to input and output files, communicate with the operator's terminal, and format exception codes into messages that can be sent to the operator. This chapter discusses these system calls.

### ESTABLISHING INPUT AND OUTPUT CONNECTIONS

The Human Interface provides two system calls for establishing connections to input and output files: `C$GET$INPUT$CONNECTION` and `C$GET$OUTPUT$CONNECTION`. These system calls are structured so that you can use the output from `C$GET$INPUT$PATHNAME` and `C$GET$OUTPUT$PATHNAME` as input to these system calls.

#### USING `C$GET$INPUT$CONNECTION`

`C$GET$INPUT$CONNECTION` obtains a connection to a file and opens that connection for reading. One of the parameters of `C$GET$INPUT$CONNECTION` is a pointer to a string containing the pathname of the file for which the connection is sought. This pathname can be the pathname returned by `C$GET$INPUT$PATHNAME` or it can be the pathname of any other file to which you want a connection. If `C$GET$INPUT$CONNECTION` cannot obtain a connection to the specified file for any reason, it returns an exception code and writes a message to `:CO:` (normally the operator's terminal) to indicate the type of problem. For example, if the specified input file does not exist, `C$GET$INPUT$CONNECTION` displays the following message:

```
<pathname>, file not found
```

The system call displays similar messages in other situations. Refer to the description of `C$GET$INPUT$CONNECTION` in Chapter 7 for more information.

Because `C$GET$INPUT$CONNECTION` returns messages to the operator in the event of an exceptional condition, your command does not have to return additional messages unless you require them. The command only has to decide whether to abort or to continue with processing.

#### USING `C$GET$OUTPUT$CONNECTION`

`C$GET$OUTPUT$CONNECTION` obtains a connection to a file and opens that connection for writing. Like `C$GET$INPUT$CONNECTION`, one of the parameters of `C$GET$OUTPUT$CONNECTION` is a pointer to a string containing the pathname of the file for which a connection is sought. This pathname

## I/O AND MESSAGE PROCESSING

can be the pathname returned by `C$GET$OUTPUT$PATHNAME` or it can be the pathname of any other file to which you want a connection. There is another parameter in `C$GET$OUTPUT$CONNECTION` which specifies the type of preposition to use when writing to the output file (`TO`, `OVER`, or `AFTER`). This preposition governs how data gets written to the file.

If you specify the `TO` preposition and the pathname of an existing file, `C$GET$OUTPUT$CONNECTION` prompts the operator for permission to delete the existing file. This prompt appears as:

```
<pathname>, already exists, OVERWRITE?
```

If the operator enters a "Y" or "y", the system call obtains the connection to the existing file. If the operator enters "N" or "n", the system call returns an exception code without obtaining a connection to the file.

If you specify the `OVER` preposition, `C$GET$OUTPUT$CONNECTION` obtains the connection without prompting the operator for permission.

If you specify the `AFTER` preposition, `C$GET$OUTPUT$CONNECTION` obtains the connection without prompting the operator for permission. It also seeks to the end of file before returning control. Thus any information you write to the file will not overwrite the existing information. This is unlike `TO` and `OVER` which cause `C$GET$OUTPUT$CONNECTION` to leave the file pointer at the beginning of the file.

If the operator does not have the proper access rights to the file, or if for some reason `C$GET$OUTPUT$CONNECTION` cannot obtain a connection to the file, `C$GET$OUTPUT$CONNECTION` returns an exception code and displays a message at the operator's terminal. Refer to the description of `C$GET$OUTPUT$CONNECTION` in Chapter 7 for more information.

### EXAMPLE PROGRAM SCENARIO

A normal scenario for using `C$GET$INPUT$CONNECTION` and `C$GET$OUTPUT$CONNECTION` is as follows:

```
DO;
```

```
    Obtain input pathname from command line with C$GET$INPUT$PATHNAME
```

```
    Obtain output pathname from command line with  
    C$GET$OUTPUT$PATHNAME
```

```
    Obtain connection to input file with C$GET$INPUT$CONNECTION
```

```
    Obtain connection to output file with C$GET$OUTPUT$CONNECTION
```

```
    Read information from input file
```

```
    Perform command operations on information
```

Write information to output file

Delete connections to input and output files

UNTIL no more input and output pathnames remain

The program listing in Figure 3-1 shows an implementation of this scenario.

#### COMMUNICATING WITH THE OPERATOR'S TERMINAL

The Human Interface provides two system calls that ease the process of communicating with the operator's terminal. They are C\$SEND\$CO\$RESPONSE and C\$SEND\$EO\$RESPONSE. Each of these system calls combines into a single system call several operations that you would normally perform when communicating with the terminal.

In its general form, C\$SEND\$CO\$RESPONSE establishes connections to :CI: (console input) and :CO: (console output), writes a message to :CO:, and reads a message from :CI:. As input to this system call, you can specify the message to be sent, the size of the message to be received, and the buffer to receive the message. Depending on the values you choose for the parameters, you can either:

- o Send a message and receive a message
- o Send a message without waiting to receive a message
- o Receive a message without sending anything

If you use C\$SEND\$CO\$RESPONSE, you do not have to invoke other system calls to attach, open, read from, or write to the operator's terminal.

There is only one difference between C\$SEND\$CO\$RESPONSE and C\$SEND\$EO\$RESPONSE. C\$SEND\$CO\$RESPONSE deals specifically with the logical names :CI: and :CO:. Therefore, its input and output can be redirected to files by changing the pathnames represented by these logical names. This is what happens when an operator places a command in a SUBMIT file; SUBMIT assumes that :CI: is the SUBMIT file and that :CO: is the output file specified in the SUBMIT command. On the other hand, while C\$SEND\$EO\$RESPONSE performs the same operations as C\$SEND\$CO\$RESPONSE, C\$SEND\$EO\$RESPONSE always reads information from and writes information to the operator's terminal. Input and output cannot be redirected with C\$SEND\$EO\$RESPONSE.

C\$SEND\$CO\$RESPONSE and C\$SEND\$EO\$RESPONSE are especially useful if you have multiple tasks communicating with a single terminal. If a task uses either of these system calls and requests a response from the terminal, no other output is displayed at the terminal until the operator enters a response to the first system call. After the operator responds, tasks can send further information to the terminal. This mechanism prevents the operator from receiving several requests for information before being able to respond to the first one.

FORMATTING EXCEPTION CODES INTO MESSAGES

Whenever you include iRMX 86 system calls in the code of a command that you write, it is possible for those system calls to encounter exceptional conditions. Exceptional conditions are divided into two categories: programming errors and environmental conditions. Programming errors occur when the iRMX 86 Operating System detects a condition that normally can be avoided by correct coding. Environmental conditions, in contrast, are generally outside the control of the application program.

Even the most thoroughly debugged commands can encounter exceptional conditions. The exceptional conditions can arise from invalid operator entries, lack of secondary storage space, media errors, and other problems over which the command has no control. The Human Interface provides a default exception handler to handle exceptional conditions in commands that you write. This exception handler receives control on the occurrence of all exceptional conditions. It displays the exception code value and mnemonic at the operator's terminal and aborts the command.

In many cases, you might want to provide your own exception handling, either to pass additional information to the operator or to allow the operator another chance to enter correct information. In such cases, you can use the Nucleus system calls GET\$EXCEPTION\$HANDLER and SET\$EXCEPTION\$HANDLER to assign your own exception handler or to cancel the effect of the default exception handler on some or all exceptions that occur in your command. Refer to the iRMX 86 NUCLEUS REFERENCE MANUAL for more information about these system calls.

When you perform your own exception handling, you will probably create special messages that you return to the operator in the event of certain exceptional conditions. However, you might not want to create messages for all possible exception codes. For this situation, the Human Interface provides the the C\$FORMAT\$EXCEPTION system call.

C\$FORMAT\$EXCEPTION accepts an exception code value as input and returns a string whose contents describe the exceptional condition. You can use this string as input to a system call such as C\$SEND\$CO\$RESPONSE to write the information to the operator terminal. By using C\$FORMAT\$EXCEPTION, you can return a message to the operator for all exceptional conditions, but you do not have to enlarge your program by including the text of these messages in the code of your command.

The text portion of the string produced by C\$FORMAT\$EXCEPTION consists of the exception code value and mnemonic in the following format:

value : mnemonic

You can display this string as is, or you can place additional explanatory text in the string before displaying it.

\*\*\*



## CHAPTER 5. COMMAND PROCESSING

When you write your own command, you might want to perform an operation that is already provided in another command (such as copying one file to another, displaying a directory, etc.). Instead of duplicating the code for this operation in your command, you can invoke Human Interface system calls to issue the commands themselves. The Human Interface provides three system calls to facilitate this process of programmatic command invocation: `C$CREATE$COMMAND$CONNECTION`, `C$SEND$COMMAND`, and `C$DELETE$COMMAND$CONNECTION`.

Invoking commands programmatically involves the following operations:

- Creating an object (called a command connection) to store the command invocation lines
- Sending the command line to the command connection and invoking the command
- Deleting the command connection

This chapter discusses these operations and provides an example of how the iRMX 86 system calls appear in a program.

### CREATING A COMMAND CONNECTION

Before you can send a command line to the Operating System to be invoked, you must create an object (called a command connection) to store the command line. The `C$CREATE$COMMAND$CONNECTION` system call creates this object and returns a token for the command connection. The token can be used in calls to `C$SEND$COMMAND` (to send command lines to the object) and in calls to `C$DELETE$COMMAND$CONNECTION` (to delete the object after using it).

When you call `C$CREATE$COMMAND$CONNECTION`, you also specify tokens for the connections that serve as command input and command output for the invoked command. This allows you to redirect input and output for the invoked command to secondary storage files. Or you can specify the normal `:CI:` and `:CO:.`

The command connection is necessary to support the processing of multiple-line commands without interference from other tasks. If not for the command connections, the Operating System would be unable to determine which continuation line went with which command when many tasks were sending command lines to be processed. The command connection provides a place to store command lines until the command is complete.

## COMMAND PROCESSING

### SENDING COMMAND LINES TO THE COMMAND CONNECTION AND INVOKING THE COMMAND

The `C$SEND$COMMAND` system call sends command lines to a command connection and, when the command invocation is complete, invokes the command. One of the parameters of this system call is the token for a command connection, which identifies the command connection to use. Another parameter is a pointer to a string which must contain a command line. The format of the command line is the same as the format for entering the command line at a terminal. The command can be any iRMX 86 Human Interface command (as described in the iRMX 86 OPERATOR'S MANUAL) or any command that you write.

If the string specified as a parameter to `C$SEND$COMMAND` contains a complete command invocation, `C$SEND$COMMAND` places the command line in the command connection and invokes the command.

However, if the string does not contain the entire command invocation (that is, it contains the "&" as a continuation character), `C$SEND$COMMAND` places the command line in the command connection without invoking the command. It also returns a condition code informing the calling program that the command is continued. This allows the calling program to change the system prompt (if reading the command invocation from a terminal) and to make additional calls to `C$SEND$COMMAND`. Additional `C$SEND$COMMAND` calls place continuation lines in the command connection, combining them with the command lines already there. When `C$SEND$COMMAND` sends the last portion of the command invocation (a line without a continuation character), it also invokes the entire command.

Once you call `C$SEND$COMMAND` enough times to place a complete command invocation in the command connection, `C$SEND$COMMAND` invokes the command. This involves loading the command from secondary storage and starting it running. The `C$SEND$COMMAND` call that invokes the command does not return control until the invoked command finishes processing. Once the command finishes processing, you can use the command connection for invoking other commands.

The `C$SEND$COMMAND` system call contains two pointers to words that receive iRMX 86 condition codes. One of these (called `except$ptr` in the system call description) points to a word that receives the status of the `C$SEND$COMMAND` system call. An `E$OK` indicates that `C$SEND$COMMAND` received the full command invocation and invoked the command. An `E$CONTINUED` indicates that the command invocation is not complete (the last line contained a continuation character). Other exception codes indicate other problems with the system call.

The other pointer (called `command$except$ptr` in the system call description) points to a word that receives the status of the invoked command. This allows you to determine the status of the invoked command.

## COMMAND PROCESSING

### DELETING THE COMMAND CONNECTION

After you have finished invoking commands programmatically, you must delete the command connection. The C\$DELETE\$COMMAND\$CONNECTION system call performs this operation. You do not need to delete the command connection after each command invocation, because the command connection is re-usable. However, you should delete the command connection after performing all C\$SEND\$COMMAND operations. This frees the memory used by the data structures of the command connection.

### EXAMPLE

Figure 5-1 contains an example of a program that uses C\$CREATE\$COMMAND\$CONNECTION, SEND\$COMMAND, and DELETE\$COMMAND\$CONNECTION. It invokes the Human Interface COPY command programmatically.

---

```
/*
 *
 * This example demonstrates the use of the following Human Interface
 * advanced standard functions:
 *
 * rq$C$create$command$connection
 * rq$C$send$command
 * rq$C$delete$command$connection
 *
 * This program uses the previous system calls to invoke the COPY
 * command from within and then continue normal processing. The
 * program is invoked with the command line:
 *
 *      PROG2
 */
*****/

prog2: DO;

$include (hexcep.lit)
$include (hcrccn.ext)
$include (hsndcmd.ext)
$include (hdlccn.ext)
$include (iexioj.ext)
$include (hgtincn.ext)
$include (hgtoen.ext)

DECLARE (ci$token, co$token, command$connection$token) WORD,
        (excep, comexcep, exexcep) WORD;
DECLARE output$prep BYTE;
```

Figure 5-1. Command Connection Example

COMMAND PROCESSING

---

```
      .  
      .  
      .  
/* Invoke utility to copy file OLD to file NEW */  
  
/* Get tokens for CI and CO */  
ci$token = rq$C$get$input$connection(@ (4, ':CI:'), @excep);  
IF excep <> E$OK THEN  
    CALL rq$exit$io$job (excep, 0, exexcep);  
co$token = rq$C$get$output$connection(@ (4, ':CO:'), output$prep, @excep);  
IF excep <> E$OK THEN  
    CALL rq$exit$io$job (excep, 0, exexcep);  
  
/* Create command connection */  
command$connection$tok = rq$C$create$command$connection (@ci$token,  
                                                         co$token, 0,  
                                                         @excep);  
  
/* Send command to copy files */  
CALL rq$C$send$command (command$connection$tok,  
                       @(23, 'COPY :F1:OLD TO :F1:NEW'),  
                       @comexcep, @excep);  
IF excep <> E$OK THEN  
    CALL rq$exit$io$job (excep, 0, exexcep);  
  
/* Delete command connection */  
CALL rq$C$delete$command$connection (command$connection$tok, @excep);  
IF excep <> E$OK THEN  
    CALL rq$exit$io$job (excep, 0, exexcep);  
  
      .  
      .      Rest of program  
      .  
  
/* Finish I/O processing */  
CALL rq$exit$io$job (excep, 0, @exexcep);  
  
END prog2;
```

Figure 5-1. Command Connection Example (continued)

---

## CHAPTER 6. PROGRAM CONTROL

Normally, when a Human Interface command is executing, an operator cannot communicate with the command (or with the application system in general) unless the command initiates the communication by requesting input from the terminal. This can present problems if an operator inadvertently enters the wrong command, or if the operator decides while the command is executing that the command is unnecessary. Under these circumstances, the operator can enter a Control-C character. In the default case, the Control-C causes the Human Interface to abort the currently-executing command. However, you can override the default Control-C mechanism by providing your own code to process Control-C characters. This chapter discusses how to do this.

### HOW THE DEFAULT CONTROL-C MECHANISM WORKS

When the operator enters a Control-C, the Operating System sends a unit to a semaphore. In the default case, it sends the unit to a semaphore established by the Human Interface. A Human Interface task waits at that semaphore to receive the unit. When it receives the unit, it aborts the command that is currently executing and returns control to the operator at the Human Interface level. The Human Interface task then waits at the semaphore for another unit.

This Control-C facility allows operators to cancel commands while the commands are executing. It is a valuable facility that can be used with your commands without requiring you to provide special implementation code.

### PROVIDING YOUR OWN CONTROL-C MECHANISM

With some commands that you write, you might want to override the default Control-C mechanism. For example, suppose you write a text editor. An operator invokes the editor with a Human Interface command and then specifies edit commands to enter text into a buffer and modify that text. While using the editor, the operator does not want a Control-C character to abort the entire editing session, destroying text in the editing buffer that may have taken an hour or more to create. Instead, the operator might want a Control-C to abort an individual editor command, but not abort the entire editor. In order to provide this facility, your Human Interface command (the editor) must override the default Control-C mechanism and provide its own code to handle Control-C entries.

To override the default Control-C mechanism, you must change the semaphore to which the Operating System sends the unit when the operator enters a Control-C. By changing the semaphore to one that you create, you circumvent the Control-C task of the Human Interface.

## PROGRAM CONTROL

You can use the \$\$\$SPECIAL system call of the Extended I/O System to change the Control-C semaphore. This system call is described in the iRMX 86 EXTENDED I/O SYSTEM REFERENCE MANUAL. However, it has three parameters that are important when changing the semaphore. They are:

connection	This parameter should contain the token for a connection to the operator's terminal.
function	This parameter should contain the value 6 to indicate the set signal character function.
data\$ptr	This parameter should point to a structure of the following form:

```
DECLARE signal$pair STRUCTURE(  
    semaphore      WORD,  
    character      BYTE);
```

where:

semaphore	A token for your new Control-C semaphore.
character	The character code for the Control-C character. If you use the ASCII code (03), the Operating System will place a unit in the semaphore when an operator enters Control-C. If you use the ASCII code plus 20H (23H), the Operating System clears out the terminal's input buffers in addition to placing the unit in the semaphore.

If your command switches the Control-C semaphore, it must also service that semaphore. It can do this either by creating a task that waits continually at the semaphore for a unit or by containing in-line code that periodically checks the semaphore.

In either case, when a unit is sent to the semaphore, the command (or the task) must perform the necessary Control-C operation.

The program flow of such a command would be:

1. Call CREATE\$SEMAPHORE to create the Control-C semaphore.
2. If you plan to create a Control-C task to service the semaphore, call CATALOG\$OBJECT to catalog the token for the semaphore in an object directory.
3. Call \$\$ATTACH\$FILE to obtain a connection to the terminal. Use logical name :CI: as the pathname parameter.
4. Call \$\$OPEN to open the connection to the terminal for reading only (mode 1).

## PROGRAM CONTROL

5. If you plan to use a Control-C task, call `CREATE$TASK` to start the Control-C task.
6. Call `$$$SPECIAL` to switch the Control-C semaphore to the one just created. Use the token for the connection to the terminal as input.
7. Continue with command processing. If you are servicing the Control-C semaphore in-line, periodically check the semaphore (by calling `RECEIVE$UNITS`) to determine if it contains any units. If you obtain a unit from the semaphore, perform the necessary Control-C processing.

If you service the Control-C with a task, the program flow of the Control-C task would be:

1. Call `LOOKUP$OBJECT` to obtain the token for the semaphore.
2. Do forever:
  - a. Call `RECEIVE$UNITS` to obtain a unit from the semaphore.
  - b. Perform the operation that must occur when the operator enters a Control-C.

Each method of servicing the Control-C semaphore has advantages and disadvantages.

If you service the Control-C semaphore with in-line code, you can perform any operation that you want. You can branch to various locations, you can start new tasks running, you can abort the command, or you can perform any other function that you wish. However, in order to service the Control-C semaphore with in-line code, you must check the semaphore periodically, to see if it contains a unit. When doing this, you must ensure that you place the checks inside all program loops that perform operations an operator might want to abort. Also, because you can check the semaphore only periodically, you cannot guarantee a quick response to the Control-C in all cases.

If you use a Control-C task, you can guarantee quick service because the task is always waiting at the semaphore. However, because a separate task services the Control-C, you can perform only a limited number of operations in response to the Control-C.

- The task can send a message to the command, but then the command would have to periodically check a mailbox. This has the same disadvantages as in-line servicing with none of the advantages.
- The task can delete the command. However, the task has no way of knowing what operations the command was performing when the operator entered the Control-C. If the command was updating an internal table, deleting the command could corrupt your entire system.

## PROGRAM CONTROL

Therefore, unless you have a specific reason for using a Control-C task, this manual recommends that you use in-line code to service the Control-C semaphore.

\*\*\*



## CHAPTER 7. CREATING HUMAN INTERFACE COMMANDS

This chapter discusses the steps that you must perform to create your own Human Interface commands. It discusses the necessary elements of a command as well as how to compile (or assemble) and link your code.

To perform the operations described in this chapter you must have either an iAPX 86-based Microcomputer Development System (such as a Series III) or an iRMX 86-based system that includes the Human Interface commands. Either system must have an editor, the necessary compiler or assembler, and the utility programs (such as LINK86).

### ELEMENTS OF A HUMAN INTERFACE COMMAND

This section discusses the rules that every command you write must obey. It also suggests some programming practices to make coding and using your command easier.

### PARSING THE COMMAND LINE

If you are going to allow the operator to enter parameters when invoking the command, the first thing your command should do is parse the command line. Chapter 3 describes the Human Interface system calls that you can use for this. To support lists of pathnames and wild-carded pathnames, the flow of a program that uses input and output files should be:

1. Call C\$GET\$INPUT\$PATHNAME to obtain the first input pathname.
2. Call C\$GET\$OUTPUT\$PATHNAME to obtain the preposition and first output pathname.
3. Call C\$GET\$PARAMETER as many times as necessary to get all the parameters.
4. Do until no more input pathnames remain:
  - a. Call C\$GET\$INPUT\$CONNECTION to obtain a connection to the input file.
  - b. Call C\$GET\$OUTPUT\$CONNECTION to obtain a connection to the output file.
  - c. Read the information from the input file, perform the command operations based on that input, and write information to the output file.

## CREATING HUMAN INTERFACE COMMANDS

- d. Call `S$DELETE$CONNECTION` (Extended I/O System call) to delete the connections to the input and output files.
- e. Call `C$GET$INPUT$PATHNAME` and `C$GET$OUTPUT$PATHNAME` to obtain the next input and output pathnames.

### AVOIDING THE USE OF CERTAIN SYSTEM CALLS

When you write the code for your Human Interface command, you can use any of the iRMX 86 system calls, depending on the requirements of your command. However, some system calls are intended primarily for use in system-level jobs (those jobs that you configure into the Operating System rather than invoking as Human Interface commands). In the descriptions of system calls, the iRMX 86 reference manuals contain cautions concerning those system calls that you should avoid using.

In particular, avoid iRMX 86 objects (and their associated system calls) that, by their use, make your command immune to deletion. Regions and extension objects (described in the iRMX 86 NUCLEUS REFERENCE MANUAL) are examples of such objects. If your command becomes immune to deletion, a Control-C that an operator enters to cancel the command will have no effect; also the operator's terminal may deadlock when the command completes processing.

### TERMINATING THE COMMAND

When the operator invokes a command, the Operating System loads the command into memory and creates an I/O job as the environment in which the command runs. (The iRMX 86 EXTENDED I/O SYSTEM REFERENCE MANUAL discusses I/O jobs.) Until the command finishes processing, the operator is unable to run any other commands. In order to finish processing correctly, any task in the command that exits must do so by calling `EXIT$IO$JOB` (an Extended I/O System call, described in the iRMX 86 EXTENDED I/O SYSTEM REFERENCE MANUAL). This system call causes the Operating System to delete the I/O job containing the command, therefore returning control to the operator. If the command omits the call to `EXIT$IO$JOB`, the operator might not be able to enter further commands.

### INCLUDE FILES

When you write the source code for your commands, you can use `$INCLUDE` statements to include the following kinds of information: external declarations of system calls, literal definitions of exception codes, and common pieces of code that you declare.

## CREATING HUMAN INTERFACE COMMANDS

As part of writing the code for your commands, you must declare each iRMX 86 system call as an external procedure. Instead of writing this code yourself, you can use the \$INCLUDE statement to include this information from files on one of the iRMX 86 release diskettes. This diskette contains a file for each system call, with the external declaration of that system call as the contents of the file. To use these files, simply determine the system calls that your command uses and place into your source code \$INCLUDE statements for the corresponding external declaration files.

You also require literal definitions of exception codes so that you can refer to the exception codes by their mnemonics instead of by their values (for example, E\$MEM instead of 2H). The Include Files release diskette contains several files (one for each layer of the Operating System) consisting of LITERALLY statements. Each file defines all the iRMX 86 condition code mnemonics used in that layer. You should copy these files, delete entries if you can guarantee that the deleted exception codes will never appear, and use \$INCLUDE statements to include them in the compilation of your command.

Refer to the iRMX 86 INSTALLATION GUIDE for information about the release diskettes and the files contained in them. Refer to the PL/M-86 USER'S GUIDE for information about the \$INCLUDE statement.

### PRODUCING AN EXECUTABLE COMMAND

After you have written the source code for your command, you must produce object code that can be executed in an iRMX 86 environment. This involves the following procedure:

1. Compile (or assemble) the command using the appropriate translators. When you do this, ensure that the names you specify in \$INCLUDE statements specify the correct devices and directories.
2. Using LINK86, link the code to iRMX 86 interface libraries (and any other libraries that you require) and produce a relocatable object module that the Operating System can load anywhere in memory. The format of the LINK86 command is:

```
LINK86                                &
    command-pathname,                 &
    :dir:HPIFC.LIB,                    &
    :dir:LPIFC.LIB,                    &
    :dir:EPIFC.LIB,                    &
    :dir:IPIFC.LIB,                    &
    :dir:RPIFC.LIB,                    &
    :dir:other.lib                     &
TO   output-pathname                  &
PRINT(mapfile-pathname) SYMBOLCOLUMNS(2) &
OBJECTCONTROLS(PURGE)                 &
BIND SEGSIZE(STACK(stacksize)) MEMPOOL(minsize,maxsize)
```

## CREATING HUMAN INTERFACE COMMANDS

where:

command-pathname	Complete pathname of the file containing your compiled (or assembled) command. You can link in several files or libraries at this point, if necessary.
other.lib	Any other files or libraries that you need to link with your command.
output-pathname	Complete pathname of the file in which LINK86 places the linked command.
mapfile-pathname	Complete pathname of the file on which LINK86 places the link map.
stacksize	Size, in bytes, of the stack needed by the command and any system calls that the command makes. The Human Interface uses this value when it creates a job for the command. Be sure the stack is large enough to handle both user and system requirements. Refer to the iRMX 86 PROGRAMMING TECHNIQUES manual for information about stack requirements of the system calls.
minsize maxsize	Minimum and maximum amount of dynamic memory, in bytes, required by the command. The command uses this memory when it creates iRMX 86 objects. The Human Interface uses the minsize and maxsize values when it creates a job for the command. Be sure that these values are large enough to satisfy the needs of your command and small enough to allow the command to be loaded into the operator's memory partition.

This command produces relocatable code that the Operating System can load into any available memory. If you require your command to be available as absolute code, you can use LINK86 and LOC86 to produce this code. Refer to the iAPX 86, 88 FAMILY UTILITIES USER'S GUIDE for more information about LINK86 and LOC86. If you require absolute code for your commands, you must also configure the Operating System in such a way that it reserves the memory locations required by the command. If it does not, the command, when loaded into the system, could overwrite Operating System or user information. Refer to the iRMX 86 CONFIGURATION GUIDE for more information about Operating System configuration.

If you are using an iRMX 86-based system to compile and link your command, the command is now ready for execution. An operator can invoke the command by entering the pathname of the file containing the linked command (the output-pathname in the LINK86 command).

## CREATING HUMAN INTERFACE COMMANDS

If you are using a Microcomputer Development System to compile or link your command, you must connect the development system to your iRMX 86 application system via the iSBC 957B package and use the Human Interface UPCOPY command to copy the linked command from the development system disk to an iRMX 86 secondary storage device. The iSBC 957B package is described in the USER'S GUIDE FOR THE iSBC 957B iAPX 86,88 INTERFACE AND EXECUTION PACKAGE. The UPCOPY command is described in the iRMX 86 OPERATOR'S MANUAL. After you transfer the linked command to an iRMX 86 secondary storage device, an operator can invoke the command by entering its pathname.

\*\*\*



## CHAPTER 8. HUMAN INTERFACE SYSTEM CALLS

The Human Interface system calls described in this chapter are presented in alphabetical sequence without regard to functional organization. A functional grouping of the calls according to type is provided in the System Call Dictionary in Table 8-1. For each call, the information is organized into the following categories:

- Brief functional description.
- Calling sequence format.
- Input parameter definitions, if applicable.
- Output parameter definitions, if applicable.
- Considerations and consequences of call usage.
- Potential exception codes, and their possible causes.

This chapter refers to PL/M-86 data types such as BYTE, WORD, and SELECTOR and iRMX 86 data types such as STRING. These words, when used as data types, are always capitalized; their definitions are found in Appendix A. This chapter also refers to an iRMX 86 data type called TOKEN. If your compiler supports the SELECTOR data type, you can declare a TOKEN to be literally a SELECTOR or a WORD. The word "token" in lower case refers to a value that the iRMX 86 Operating System assigns to an object. The Operating System returns this value to a TOKEN (the data type) when it creates the object.

If you are a new user of the Human Interface calls, it is suggested that you review the parsing considerations in Chapter 3 before writing your source code. You should also review the format of the released Human Interface commands. They are described in the iRMX 86 OPERATOR'S MANUAL.

This chapter assumes that you are familiar with a number of terms and concepts that are common to the iRMX 86 Operating System. If you are not, you should read INTRODUCTION TO THE iRMX 86 OPERATING SYSTEM and the chapters in the iRMX 86 NUCLEUS REFERENCE MANUAL that refer to the terms "memory pool" and "catalog."

HUMAN INTERFACE SYSTEM CALLS

Table 8-1. System Call Dictionary

System Call	Synopsis	Page
I/O Processing Calls		
C\$GET\$INPUT\$CONNECTION	Return an EIOS connection for the specified input file.	8-15
C\$GET\$OUTPUT\$CONNECTION	Return an EIOS connection for the specified output file.	8-25
Command Parsing Calls		
C\$GET\$CHAR	Get a character from the command line	8-11
C\$GET\$INPUT\$PATHNAME	Parse the command line and return an input pathname.	8-20
C\$GET\$PARAMETER	Parse the command line for the next parameter and return it as a keyword name and a value.	8-34
C\$GET\$OUTPUT\$PATHNAME	Parse the command line and return an output pathname.	8-31
C\$SET\$PARSE\$BUFFER	Parse a buffer other than the current command line.	8-51
C\$GET\$COMMAND\$NAME	Return the command name by which the the current command was invoked	8-13
Message Processing Calls		
C\$FORMAT\$EXCEPTION	Create a default message for an exception code and place it in a user buffer.	8-9
C\$SEND\$CO\$RESPONSE	Send a message to the command output (CO) and read a response from the command input (CI).	8-45
C\$SEND\$EO\$RESPONSE	Send a message to the operator's terminal and return a response from that terminal.	8-48



HUMAN INTERFACE SYSTEM CALLS

Table 8-1. System Call Dictionary (continued)

System Call	Synopsis	Page
Command Processing Calls		
C\$CREATE\$COMMAND\$CONNECTION	Create a command connection and return a token.	8-4
C\$DELETE\$COMMAND\$CONNECTION	Delete a specific command connection.	8-8
C\$SEND\$COMMAND	Concatenate command lines into the data structure created by CREATE\$COMMAND\$CONNECTION and then invoke the command.	8-38

## C\$CREATE\$COMMAND\$CONNECTION

C\$CREATE\$COMMAND\$CONNECTION, a command processing call, creates an iRMX 86 object called a command connection that is required in order to invoke commands programmatically.

```
command$conn = RQ$C$CREATE$COMMAND$CONNECTION(default$ci, default$co,
                                               reserved$word,
                                               except$ptr);
```

## INPUT PARAMETERS

default\$ci	A TOKEN for a connection that is used as the :CI: (console input) for any commands you invoke using this command connection.
default\$co	A TOKEN for a connection that is used as the :CO: for any commands you invoke using this command connection.
reserved\$word	A WORD reserved for future use. Its value should be zero (0).

## OUTPUT PARAMETERS

command\$conn	A TOKEN which receives a token for the new command connection.
except\$ptr	A POINTER to a WORD in which the Human Interface returns a condition code.

## DESCRIPTION

You can use this call when you want to invoke a command programmatically instead of interactively. It provides a place to store command lines until the command invocation is complete.

The call creates an iRMX 86 object called a command connection and returns a token for that command connection. The C\$SEND\$COMMAND system call can use this token to send command lines to the command connection, where they are stored until the command invocation is complete. The command connection also defines default :CI: and :CO: connections that are used by any commands invoked via this command connection.

Although a job can contain multiple command connections, the tasks in a job cannot create command connections simultaneously. Attempts to do this result in an E\$CONTEXT exception code. Therefore, it is advisable for one task to create the command connections for all tasks in the job.

## EXCEPTION CODES

E\$OK	No exceptional conditions were encountered.
E\$CONTEXT	At least one of the following situations occurred. <ul style="list-style-type: none"> <li>● The Operating System detected two command connections being created simultaneously by two tasks in the same job. This condition occurred because a programmer miscalculated or disrupted a synchronized use of the command connection.</li> <li>● The Operating System detected the :STREAM: device, the default\$ci device, or the default\$co device in the process of being detached.</li> <li>● The job containing the task which invoked this system call was not an I/O job. (Refer to the iRMX 86 EXTENDED I/O SYSTEM REFERENCE MANUAL for information about I/O jobs.)</li> <li>● While creating a STREAM file, the Extended I/O System was unable to attach the :STREAM: device because another task had already invoked a Basic I/O system call to attach the :STREAM: device.</li> </ul>
E\$DEVFD	The Extended I/O System attempted the physical attachment of the :STREAM: device. This device had formerly been only logically attached. In the process, the Extended I/O System found that the device and the device driver specified in the logical attachment were incompatible. The Operating System would not have returned this exception code if the :STREAM: device had been properly configured in the Extended I/O and/or the Basic I/O Systems.
E\$FNEXIST	The :STREAM: file does not exist or is marked for deletion.
E\$IFDR	The Extended I/O System attempted to obtain information about the default\$ci or default\$co connection. However, the request for information resulted in an invalid file driver request.

- E\$IOMEM            The Basic I/O System job does not currently have a block of memory large enough to allow the Human Interface to create a stream file.
- E\$LIMIT            At least one of the following situations occurred.
- The Operating System detected that the object directory of the calling task's job has already reached the maximum object directory size.
  - The Operating System detected that the calling task's job has exceeded its object limit.
  - The calling task's job (or that job's default user object) is currently involved in more than 255 (decimal) I/O operations.
  - The job containing the task which invoked this call was not an I/O job. (Refer to the IRMX 86 EXTENDED I/O SYSTEM REFERENCE MANUAL for information about I/O jobs.)
- E\$LOG\$NAME  
  \$NEXIST            The Extended I/O System was unable to find the logical name :STREAM: in the object directories of the local job, the global job, and the root job.
- E\$MEM              The memory pool of the job whose task invoked this call does not currently have a block of memory large enough to allow this system call to run to completion.
- E\$NO\$PREFIX        The calling task's job does not have a valid default prefix.
- E\$NOT\$CONNECTION    The token specified in either the default\$ci or default\$co parameter does not refer to a valid connection.
- E\$NOT\$LOG\$NAME     The logical name :STREAM: does not refer to a valid connection.
- E\$NO\$USER           At least one of the following situations occurred.
- The calling task's job does not have a default user.
  - The calling task's job has a default user, but it is not a user object.

E\$PARAM

The system call forced the Extended I/O System to attempt the physical attachment of the :STREAM: device. The device had formerly been only logically attached. In the process, the Extended I/O System found that the stream file driver is not configured into your system. Hence the physical attachment is not possible.

E\$SUPPORT

Either the default\$ci or default\$co connection was not created by a task in the calling task's job.

**C\$DELETES\$COMMAND\$CONNECTION**

**C\$DELETES\$COMMAND\$CONNECTION**, a command processing call, deletes a command connection object and frees the memory used by the command connection's data structures.

<b>CALL RQ\$C\$DELETES\$COMMAND\$CONNECTION(command\$conn, except\$ptr);</b>
--

**INPUT PARAMETER**

<b>command\$conn</b>	A TOKEN for a valid command connection.
----------------------	---

**OUTPUT PARAMETER**

<b>except\$ptr</b>	A POINTER to a WORD in which the Human Interface returns a condition code.
--------------------	--

**DESCRIPTION**

This call deletes a command connection object previously defined in a **C\$CREATE\$COMMAND\$CONNECTION** call and releases the memory used by the command connection's data structures.

**EXCEPTION CODES**

<b>E\$OK</b>	No exceptional conditions were encountered.
<b>E\$EXIST</b>	The command connection parameter is not the token for an existing object.
<b>E\$TYPE</b>	The command connection parameter refers to an object that is not a command connection object.

## C\$FORMAT\$EXCEPTION

C\$FORMAT\$EXCEPTION, a message processing call, creates a default message for a given exception code and writes that message into a user-provided string.

```
CALL RQ$C$FORMAT$EXCEPTION(buff$p, buff$max, exception$code,
                           reserved$byte, except$ptr);
```

## INPUT PARAMETERS

buff\$max	A WORD that specifies the maximum number of bytes that may be contained in the string pointed to by buff\$p.
exception\$code	A WORD containing the exception code value for which a message is to be created.
reserved\$byte	A BYTE reserved for future use. Its value must be one (1).

## OUTPUT PARAMETER

buff\$p	A POINTER to a STRING into which the Human Interface concatenates the formatted exception message.
except\$ptr	A POINTER to a WORD in which the Human Interface returns a condition code.

## DESCRIPTION

C\$FORMAT\$EXCEPTION causes the Human Interface to create a message for the exception code. The message consists of the exception code value and exception code mnemonic in the following format:

```
value : mnemonic
```

where the mnemonics are provided by the Human Interface from an internal table and are listed in Appendix B of this manual.

The call concatenates the message to the end of the string pointed to by the buff\$p pointer and updates the count byte to reflect the addition. If a string is not already present in the buffer, the first byte of the buffer must be a zero. The message added by C\$FORMAT\$EXCEPTION will not be longer than 30 characters (not including the length byte).

## EXCEPTION CODES

E\$OK	No exceptional conditions were encountered.
E\$PARAM	An unknown exception code value was given.
E\$STRING	The message to be returned exceeds the length limit of 255 characters.
E\$STRING\$BUFFER	The buffer pointed to by the buff\$p parameter is not large enough to contain the exception message.



## C\$GET\$CHAR

C\$GET\$CHAR, a command parsing call, gets a character from the parsing buffer.

```
char = RQ$C$GET$CHAR(except$ptr);
```

## OUTPUT PARAMETERS

char	A BYTE in which the Human Interface places the next character of the parsing buffer. A null (00H) character is returned when parsing buffer's pointer is at the end of the buffer.
except\$ptr	A POINTER to a WORD in which the Human Interface returns a condition code.

## DESCRIPTION

When an operator invokes a command, the command's parameters are placed in a parsing buffer. The C\$GET\$CHAR system call gets a single character from that buffer and moves the parsing pointer to the next character. Consecutive calls to C\$GET\$CHAR return consecutive characters from the parsing buffer.

## EXCEPTION CODES

E\$OK	No exceptional conditions were encountered.
E\$CONTEXT	The Operating System detected a zero value for the object directory size. This indicates that your task's job is not an I/O job. Refer to the IRMX 86 EXTENDED I/O SYSTEM REFERENCE MANUAL for information about I/O jobs.
E\$LIMIT	At least one of the following situations occurred. <ul style="list-style-type: none"> <li>● The Operating System detected that the object directory of the calling task's job has already reached the maximum object directory size.</li> <li>● The Operating System detected that the calling task's job has exceeded its object limit.</li> </ul>

- The calling task's job is not an I/O job. Refer to the iRMX 86 EXTENDED I/O SYSTEM REFERENCE MANUAL for information about I/O jobs.

E\$MEM

The memory pool of the calling task's job does not currently have a block of memory large enough to allow this system call to run to completion.

## C\$GET\$COMMAND\$NAME

C\$GET\$COMMAND\$NAME, a command parsing call, obtains the pathname of the command that the operator used when invoking the command.

```
CALL RQ$C$GET$COMMAND$NAME (path$name$p, name$max, except$ptr);
```

## INPUT PARAMETER

name\$max            A WORD that specifies the length in bytes of the string pointed to by the path\$name\$p parameter.

## OUTPUT PARAMETERS

path\$name\$p        A POINTER to a STRING that receives the name of the command (the last component of the pathname).

except\$ptr         A POINTER to a WORD in which the Human Interface returns a condition code.

## DESCRIPTION

If a command needs to know the name under which it was invoked, the C\$GET\$COMMAND\$NAME returns this information. This information is available to each command and is stored in a buffer that is separate from the parsing buffer. Therefore, calling C\$GET\$COMMAND\$NAME does not obtain information from the parsing buffer, nor does it move the parsing pointer.

If the operator invokes the command without specifying a logical name, the Human Interface automatically searches a number of directories for the command. In such cases, the value returned by C\$GET\$COMMAND\$NAME also includes the directory name (such as :SYSTEM:, :PROG:, or :\$:) as a prefix to the command name.

## EXCEPTION CODES

E\$OK                No exceptional conditions were encountered.

E\$LIMIT            The calling task's job was not created by the Human Interface.

# C\$GET\$COMMAND\$NAME

E\$STRING\$BUFFER The buffer pointed to by the path\$name\$p parameter is not large enough to contain the command name.

E\$TIME The calling task's job was not created by the Human Interface.

## CSGET\$INPUT\$CONNECTION

CSGET\$INPUT\$CONNECTION, an I/O processing call, returns an Extended I/O System connection to the specified input file.

```
connection = RQ$CSGET$INPUT$CONNECTION(path$name$p, except$ptr);
```

## INPUT PARAMETER

path\$name\$p            A POINTER to a STRING containing the pathname of the file to be accessed.

## OUTPUT PARAMETERS

connection            A TOKEN in which the Operating System returns the token for the connection to the specified pathname.

except\$ptr            A POINTER to a WORD in which the Human Interface returns a condition code.

## DESCRIPTION

CSGET\$INPUT\$CONNECTION obtains a connection to the specified file. This connection is open for reading and has the following attributes:

- Read only
- Accessible to all users
- Has two 1024-byte buffers

CSGET\$INPUT\$CONNECTION causes an error message to be displayed at the operator's terminal (:CO:) whenever the Operating System encounters an exceptional condition. The exceptional condition that triggers the error message can either be one of those listed for CSGET\$INPUT\$CONNECTION or it can be one of those associated with the Extended I/O System calls S\$ATTACH\$FILE and S\$OPEN. The following messages can occur:

- <pathname>, file does not exist  
The input file does not exist.

# C\$GET\$INPUT\$CONNECTION

- <pathname>, invalid file type

The input file was a data file and a directory was required, or vice versa.

- <pathname>, invalid logical name

The input pathname contains a logical name that is longer than 12 characters, that contains unmatched colons, or that contains invalid characters.

- <pathname>, logical name does not exist

The input pathname contains a logical name that does not exist.

- <pathname>, READ access required

The user does not have read access to the input file.

- <pathname>, <exception value>:<exception mnemonic>

An exceptional condition occurred when C\$GET\$INPUT\$CONNECTION attempted to obtain the input connection. The <exception value> and <exception mnemonic> portions of the message indicate the exception code encountered. Refer to "Exception Codes" in this call description and to the descriptions of S\$ATTACH\$FILE and S\$OPEN in the IRMX 86 EXTENDED I/O SYSTEM REFERENCE MANUAL.

## EXCEPTION CODES

- |            |   |
|------------|---|
| E\$OK      | No exceptional conditions were encountered.   |
| E\$CONTEXT | At least one of the following situations occurred. <ul style="list-style-type: none"><li>● The device specified in the path\$name\$p parameter is in the process of being detached.</li><li>● The calling task's job is not an I/O job. (Refer to the IRMX 86 EXTENDED I/O SYSTEM REFERENCE MANUAL for more information about I/O jobs.)</li><li>● The Extended I/O System attempted to logically attach the device containing the file specified in the path\$name\$p parameter. However, the device is already attached.</li><li>● The Extended I/O System attempted to attach a stream file and in so doing issued an invalid stream file request.</li></ul> |

- E\$DEVFD** The Extended I/O System attempted the physical attachment of a device that had formerly been only logically attached. In the process, the Extended I/O System found that the device and the device driver specified in the logical attachment were incompatible. The Operating System would not have returned this exception code if the device referred to by the path\$name\$p parameter had been properly configured in the Extended I/O and/or the Basic I/O Systems.
- E\$EXIST** At least one of the following situations occurred.
- When attempting to obtain a connection to a file, the Operating System discovered that the connection to the device containing that file is invalid.
  - The calling task's job was not created by the Human Interface.
- E\$FACCESS** The access rights embedded in the connection have prohibited you from opening the file in the read mode. This exceptional condition can arise only when the connection refers to named files.
- E\$FNEXIST** At least one of the following circumstances occurred.
- A file in the specified pathname (referred to by the path\$name\$p parameter), or the target file itself, does not exist or is marked for deletion. For example, in the pathname A/B/C, this exception code would be returned if A, B, or C was marked for deletion or did not exist.
  - While attaching the file pointed to by the path\$name\$p parameter, the Extended I/O System attempted the physical attachment of the device as a named device. (This device had formerly been only logically attached.) It could not complete this process because the device's physical name does not exist.
- E\$FTYPE** The Operating System detected an error in the pathname specified by the path\$name\$p parameter. The pathname included the name of a data file as a directory. For example, the pathname A/B/C assumes that A and B are names for directories. This exception code would have been returned if either A or B was actually a data file.

E\$ILLVOL	While attaching the file pointed to by the path\$name\$p parameter, the Extended I/O System attempted the physical attachment of the device as a named device. This device had formerly been only logically attached. In the process, the Extended I/O System examined the volume label and found that the volume did not contain named files. This prevented the Extended I/O System from completing physical attachment because the named file driver was requested during logical attachment.
E\$IO\$HARD	While attempting to access the file specified in the path\$name\$p parameter, the Operating System detected a hard I/O error.
E\$IOMEM	While attempting to create a connection, memory from the Basic I/O subsystem's memory pool was needed. However, the Basic I/O System job does not currently have a block of memory large enough to allow this system call to run to completion.
E\$IO\$OPRINT	While attempting to access the file specified in the path\$name\$p parameter, the Operating System detected that the device was off-line. Operator intervention is required. C\$FORMAT\$EXCEPTION returns the value E\$IO\$NOT\$READY for this code.
E\$IO\$SOFT	While attempting to access the file specified in the path\$name\$p parameter, the Operating System detected a soft I/O error. It tried the operation again but was unsuccessful.
E\$IO\$UNCLASS	An unknown type of I/O error occurred while trying to access the file given in the path\$name\$p parameter.
E\$LIMIT	At least one of the following situations occurred. <ul style="list-style-type: none"> <li>● The Operating System detected either the calling task's job or the job's default user object as being involved in more than 255 (decimal) I/O operations.</li> <li>● The calling task's job was not created by the Human Interface.</li> </ul>
E\$LOG\$NAME\$- NEXIST	The pathname for the specified device (referred to by the path\$name\$p parameter) contains an explicit logical name. The Extended I/O System was unable to find this name in the object directories of the local job, the global job, and the root job.
E\$LOG\$NAME\$- SYNTAX	The pathname pointed to by the path\$name\$p parameter contains a logical name. However, the logical name contains unmatched colons, is longer than 12 characters, or contains invalid characters.



E\$MEDIA	The Operating System detected that the device containing the specified file (referred to by the path\$name\$p parameter) was not online.
E\$MEM	The memory pool of the calling task's job does not currently have a block of memory large enough to allow this system call to run to completion.
E\$NO\$PREFIX	The pathname specified in the path\$name\$p parameter of this call contained no explicit prefix (no logical name), so the Extended I/O System attempted to use the default prefix. However, the default prefix is either undefined, or it is not a valid device connection or file connection.
E\$NOT\$LOG\$NAME	The pathname specified in the path\$name\$p parameter contains an explicit logical name. This logical name does not refer to a valid connection.
E\$NO\$USER	The job containing the task which invoked this call does not have a default user or the default user of this calling task's job was not a user object.
E\$PARAM	At least one of the following situations occurred. <ul style="list-style-type: none"> <li>● The system call forced the Extended I/O System to attempt the physical attachment of the device referenced by the path\$name\$p parameter. The device had formerly been only logically attached. In the process, the Extended I/O System found that the logical attachment referred to a file driver (named, physical, or stream) that is not configured into your system. Hence the physical attachment is not possible.</li> <li>● The connection to the specified file cannot be opened for reading.</li> </ul>
E\$SHARE	The Operating System detected another task using the I/O System to manipulate the file through another connection. That task requested that the I/O System restrict the sharing of the file to certain modes. Your task is not allowed to read the file.

## C\$GET\$INPUT\$PATHNAME

C\$GET\$INPUT\$PATHNAME, a command parsing call, gets a pathname from the list of input pathnames in the parsing buffer.

```
CALL RQ$C$GET$INPUT$PATHNAME(path$name$p, path$name$max, except$ptr);
```

## INPUT PARAMETER

**path\$name\$max** A WORD that specifies the length in bytes of the string pointed to by the path\$name\$p parameter. The maximum length that you can specify is 256 bytes (255 characters for the pathname and one byte for the count).

## OUTPUT PARAMETERS

**path\$name\$p** A POINTER to a STRING which receives the next pathname in the pathname list. A zero-length string indicates that there are no more pathnames.

**except\$ptr** A POINTER to a WORD in which the Human Interface returns a condition code.

## DESCRIPTION

The first call to C\$GET\$INPUT\$PATHNAME retrieves the entire input pathname list and moves the parsing pointer to the next parameter. C\$GET\$INPUT\$PATHNAME stores the list in an internal buffer and returns the first pathname to the string pointed to by the path\$name\$p parameter. Succeeding calls to C\$GET\$INPUT\$PATHNAME return additional pathnames from the input pathname list but do not move the parsing pointer. C\$GET\$INPUT\$PATHNAME denotes the end of the pathname list by returning a zero-length string.

C\$GET\$INPUT\$PATHNAME accepts wild-card characters in the last component of a pathname. It treats a wild-carded pathname as a list of pathnames. To obtain each pathname, it searches in the parent directory of the wild-carded component, comparing the wild-carded name with the names of all files in the directory. It returns the next pathname that matches.

The pathname returned by C\$GET\$INPUT\$PATHNAME can be used for any purpose. However, it is most often used in a call to C\$GET\$INPUT\$CONNECTION, to obtain a connection.

## EXCEPTION CODES

- E\$OK** No exceptional conditions were encountered.
- E\$CONTEXT** At least one of the following situations occurred.
- The device pointed to by the path\$name\$p parameter is in the process of being detached.
  - The calling task's job is not an I/O job. (Refer to the iRMX 86 EXTENDED I/O SYSTEM REFERENCE MANUAL for more information about I/O jobs.)
  - The Extended I/O System attempted to logically attach the device containing the file pointed to by the path\$name\$p parameter. However, the device is already attached.
  - The Extended I/O System attempted to attach a stream file and in so doing issued an invalid stream file request.
  - The task called C\$GET\$OUTPUT\$PATHNAME before calling C\$GET\$INPUT\$PATHNAME.
- E\$DEVFD** The Extended I/O System attempted the physical attachment of a device that had formerly been only logically attached. In the process, the Extended I/O System found that the device and the device driver specified in the logical attachment were incompatible. The Operating System would not have returned this exception code if the device referred to by the path\$name\$p parameter had been properly configured in the Extended I/O and/or the Basic I/O subsystems.
- E\$EXIST** At least one of the following situations occurred.
- When attempting to obtain a connection to the parent directory of the file pointed to by the path\$name\$p parameter, the Operating System discovered that the connection to the device containing that directory is invalid.
  - The calling task's job was not created by the Human Interface.
- E\$FACCESS** The directory's access rights have prohibited you from opening the directory in the read mode.
- E\$FLUSHING** The device containing the directory was in the process of being detached.

## E\$FNEXIST

At least one of the following circumstances occurred.

- A file in the specified pathname (pointed to by the path\$name\$p parameter), or the target file itself, does not exist or is marked for deletion. For example, in the pathname A/B/C, this exception code would be returned if A, B, or C was marked for deletion or did not exist.
- While attaching the parent directory of the file pointed to by the path\$name\$p parameter, the Extended I/O System attempted the physical attachment of the device as a named device. (This device had formerly been only logically attached.) It could not complete this process because the device's physical name does not exist.

## E\$FTYPE

The Operating System detected an error in the pathname pointed to by the path\$name\$p parameter. The pathname included the name of a data file as a directory. For example, the pathname A/B/C assumes that A and B are names for directories. This exception code would have been returned if either A or B was actually a data file.

## E\$IFDR

The file listed as the parent directory of the path\$name\$p file is a data file instead of a directory.

## E\$ILLVOL

While attaching the parent directory of the file pointed to by the path\$name\$p parameter, the Extended I/O System attempted the physical attachment of the device as a named device. This device had formerly been only logically attached. In the process, the Extended I/O System examined the volume label and found that the volume did not contain named files. This prevented the Extended I/O System from completing physical attachment because the named file driver was requested during logical attachment.

## E\$IO\$HARD

While attempting to access the parent directory of the file pointed to by the path\$name\$p parameter, the Operating System detected a hard I/O error.

## E\$IOMEM

While attempting to create a connection, memory from the Basic I/O subsystem's memory pool was needed. However, the Basic I/O System job does not currently have a block of memory large enough to allow this system call to run to completion.

**E\$IO\$OPRINT** While attempting to access the parent directory of the file pointed to by the path\$name\$p parameter, the Operating System detected that the device was off-line. Operator intervention is required. C\$FORMAT\$EXCEPTION returns the value E\$IO\$NOT\$READY for this code.

**E\$IO\$SOFT** While attempting to access the parent directory of the file pointed to by the path\$name\$p parameter, the Operating System detected a soft I/O error. The error also occurred during retry.

**E\$IO\$UNCLASS** An unknown type of I/O error occurred while trying to access the parent directory of the file pointed to by the path\$name\$p parameter.

**E\$LIMIT** At least one of the following situations occurred.

- The object limit of the calling task's job has been exceeded.
- The Operating System detected either the calling task's job or the job's default user object as being involved in more than 255 (decimal) I/O operations.
- The calling task's job was not created by the Human Interface.

**E\$LIST** The last value of the input pathname list is missing. For example, "ABLE,BAKER," has no value following the second comma.

**E\$LOG\$NAME\$-  
NEXIST** The pathname for the specified device (pointed to by the path\$name\$p parameter) contains an explicit logical name. The Extended I/O System was unable to find this name in the object directories of the local job, the global job, and the root job.

**E\$LOG\$NAME\$-  
SYNTAX** The pathname pointed to by the path\$name\$p parameter contains a logical name. However, the logical name contains unmatched colons, is longer than 12 characters, or contains invalid characters.

**E\$MEDIA** The Operating System detected that the device containing the specified file (pointed to by the path\$name\$p parameter) was not online.

**E\$MEM** The memory pool of the calling task's job does not currently have a block of memory large enough to allow this system call to run to completion.

E\$NO\$PREFIX	The pathname pointed to by the path\$name\$p parameter of this call contained no explicit prefix (no logical name), so the Extended I/O System attempted to use the default prefix. However, the default prefix is either undefined, or it is not a valid device connection or file connection.
E\$NOT\$LOG\$NAME	The pathname pointed to by the path\$name\$p parameter contains an explicit logical name. This logical name does not refer to a valid connection.
E\$NO\$USER	The job containing the task which invoked this call does not have a default user or the default user of this calling task's job was not a user object.
E\$PARAM	At least one of the following situations occurred. <ul style="list-style-type: none"><li>• The Extended I/O System to attempted the physical attachment of the device pointed to by the path\$name\$p parameter. The device had formerly been only logically attached. In the process, the Extended I/O System found that the logical attachment referred to a file driver (named, physical, or stream) that is not configured into your system. Hence the physical attachment is not possible.</li><li>• The connection to the parent directory cannot be opened for reading.</li></ul>
E\$PARSE\$TABLES	The Human Interface detected an error that should not occur unless someone inadvertently alters an internal table used by the Human Interface.
E\$SHARE	The Operating System detected another task using the I/O System to manipulate through another connection the parent directory of the file pointed to by the path\$name\$p parameter. That task requested that the I/O System restrict the sharing of the file to certain modes. Therefore your task cannot access the file.
E\$STRING	The pathname to be returned exceeds the length limit of 255 characters.
E\$STRING\$BUFFER	The buffer pointed to by the path\$name\$p parameter was not large enough for the pathname to be returned.
E\$SUPPORT	The Operating System attempted to read the parent directory of the pathname pointed to by the path\$name\$p parameter. However, the file driver corresponding to that directory does not support this operation.
E\$WILD\$CARD	The pathname to be returned contains an invalid wild-card specification.

## C\$GET\$OUTPUT\$CONNECTION

C\$GET\$OUTPUT\$CONNECTION, an I/O processing call, parses the command line and returns an Extended I/O System connection referring to the requested output file.

```
connection = RQ$C$GET$OUTPUT$CONNECTION(path$name$p, preposition,
                                         except$ptr);
```

## INPUT PARAMETERS

**path\$name\$p** A POINTER to a STRING containing the pathname of the file to be accessed.

**preposition** A BYTE that defines which preposition to use to create the output file. Use one of the following values to specify the preposition mode:

<u>Value</u>	<u>Meaning</u>
0	Use same preposition as was returned by the last GET\$OUTPUT\$PATHNAME call
1	TO
2	OVER
3	AFTER
4-255	Undefined, results in an error

## OUTPUT PARAMETERS

**connection** A TOKEN in which the Human Interface returns a token for the connection to the output file.

**except\$ptr** A POINTER to a WORD in which the Human Interface returns a condition code.

## DESCRIPTION

C\$GET\$OUTPUT\$CONNECTION obtains a connection to the specified file. This connection is open for writing and has the following attributes:

- Write only
- Accessible to all
- Has two 1024-byte buffers

# C\$GET\$OUTPUT\$CONNECTION

If the call to C\$GET\$OUTPUT\$CONNECTION specifies the TO preposition and the output file already exists, C\$GET\$OUTPUT\$CONNECTION issues the following message to the terminal (:CO:):

<pathname>, already exists, OVERWRITE?

If the operator enters Y, y, R, or r, C\$GET\$OUTPUT\$CONNECTION returns a connection to the existing file, allowing the command to write over the file. Any other response causes C\$GET\$OUTPUT\$CONNECTION to skip over the pathname.

C\$GET\$OUTPUT\$CONNECTION causes an error message to be displayed at the operator's terminal (:CO:) whenever an exceptional condition occurs. The exceptional condition that triggers the error message can be either one of those listed for C\$GET\$OUTPUT\$CONNECTION or one of those associated with an Extended I/O System call. The following messages can occur:

- <pathname>, DELETE access required

The user does not have delete access to the file or directory.

- <pathname>, directory ADD entry access required

The user does not have add entry access to the directory.

- <pathname>, file does not exist

The output file does not exist,

- <pathname>, invalid file type

The output file was a data file and a directory was required, or vice versa.

- <pathname>, invalid logical name

The output pathname contains a logical name that is longer than 12 characters, that contains unmatched colons, or that contains invalid characters.

- <pathname>, logical name does not exist

The output pathname contains a logical name that does not exist.



- <pathname>, <exception value>:<exception mnemonic>

An exceptional condition occurred when C\$GET\$OUTPUT\$CONNECTION attempted to obtain the input connection. The <exception value> and <exception mnemonic> portions of the message indicate the exception code encountered. Refer to "Exception Codes" in this call description and to the IRMX 86 EXTENDED I/O SYSTEM REFERENCE MANUAL.

## EXCEPTION CODES

- |            |   |
|------------|---|
| E\$OK      | No exceptional conditions were encountered.   |
| E\$CONTEXT | At least one of the following situations occurred. <ul style="list-style-type: none"> <li>• The calling task's job was not created by the Human Interface.</li> <li>• The device referred to by the path\$name\$p parameter was in the process of being detached.</li> <li>• The Extended I/O System was unable to attach the device containing the file because the Basic I/O System has already attached the device.</li> <li>• The Extended I/O System attempted to attach a stream file and in so doing issued an invalid stream file request.</li> </ul> |
| E\$DEVFD   | The Extended I/O System attempted the physical attachment of the device referenced by the path\$name\$p parameter. This device had formerly been only logically attached. In the process, the Extended I/O System found that the device and the device driver specified in the logical attachment were incompatible. The Operating System would not have returned this exception code if the device referenced by the path\$name\$p parameter had been properly configured in the Extended I/O and/or the Basic I/O Systems.                                  |
| E\$EXIST   | When attempting to obtain a connection to a file, the Operating System discovered that the connection to the device containing that file is invalid.  |
| E\$FACCESS | At least one of the following situations occurred. <ul style="list-style-type: none"> <li>• The user did not have update access to an existing file and/or add-entry access to the parent directory.</li> <li>• The TO or OVER preposition was specified and the user did not have the ability to truncate the file.</li> </ul>   |

E\$FNEXIST	<p>At least one of the following situations occurred.</p> <ul style="list-style-type: none"> <li>• A file in the pathname specified by the path\$name\$p parameter does not exist or is marked for deletion. For example, in the pathname A/B/C, this exception code would be returned if A, B, or C was marked for deletion or did not exist.</li> <li>• While attaching the file pointed to by the path\$name\$p parameter, the Extended I/O System attempted the physical attachment of the device as a named device. (This device had formerly been only logically attached.) It could not complete this process because the device's physical name does not exist.</li> </ul>
E\$FTYPE	<p>The Operating System detected an error in the pathname specified by the path\$name\$p parameter. The pathname included the name of a data file as a directory. For example, the pathname A/B/C assumes that A and B are names for directories. This exception code would have been returned if either A or B was actually a data file.</p>
E\$IFDR	<p>The Extended I/O System attempted to obtain information about the path\$name\$p connection. However, the request for information resulted in an invalid file driver request.</p>
E\$ILLVOL	<p>The Extended I/O System attempted the physical attachment of the device referred to by the path\$name\$p parameter. (This device had formerly been only logically attached.) In the process, the Extended I/O System examined the volume label and found that the volume does not contain named files. This prevented the Extended I/O System from completing physical attachment because the named file driver was requested during logical attachment.</p>
E\$IO\$HARD	<p>While attempting to access the file specified in the path\$name\$p parameter, the Operating System detected a hard I/O error.</p>
E\$IOMEM	<p>The memory required to create a connection was part of a Basic I/O System memory pool. However, the Basic I/O System job does not currently have a block of memory large enough to allow this system call to run to completion.</p>
E\$IO\$OPRINT	<p>While attempting to access the file specified in the path\$name\$p parameter, the Operating System detected that the device was off-line. Operator intervention is required. C\$FORMAT\$EXCEPTION returns the value E\$IO\$NOT\$READY for this code.</p>

E\$IO\$SOFT	While attempting to access the file specified in the path\$name\$p parameter, the Operating System detected a soft I/O error. The error also occurred during retry.
E\$IO\$UNCLASS	An unknown type of I/O error occurred while trying to access the file given in the path\$name\$p parameter.
E\$IO\$WRPROT	While attempting to obtain an input connection to the file specified in the path\$name\$p parameter, the Operating System detected that the volume containing the file is write-protected.
E\$LIMIT	At least one of the following situations occurred. <ul style="list-style-type: none"> <li>• The Operating System detected either the calling task's job or the job's default user object as being involved in more than 255 (decimal) I/O operations.</li> <li>• The calling task's job was not an I/O job. (Refer to the IRMX 86 EXTENDED I/O SYSTEM REFERENCE MANUAL for more information about I/O jobs.)</li> </ul>
E\$LOG\$NAME\$- NEXIST	The pathname specified in the path\$name\$p parameter contains an explicit logical name. The Extended I/O System was unable to find this name in the object directories of the local job, the global job, and the root job.
E\$LOG\$NAME\$- SYNTAX	The pathname pointed to by the path\$name\$p parameter contains a logical name. However, the logical name contains unmatched colons, is longer than 12 characters, or contains invalid characters.
E\$MEDIA	The device specified by the path\$name\$p parameter was off-line.
E\$MEM	The memory pool of the calling task's job does not currently have a block of memory large enough to allow this system call to run to completion. Usually, this means the Operating System was unable to create or attach a file.
E\$NO\$PREFIX	The pathname specified by the path\$name\$p parameter of this call contained no explicit prefix (no logical name), so the Extended I/O System attempted to use the default prefix. However, the default prefix is either undefined, or it is not a valid device connection or file connection.

- E\$NOT\$LOG\$NAME** The pathname specified in the path\$name\$p parameter contains an explicit logical name. This logical name does not refer to a valid connection.
- E\$NO\$USER** The calling task's job does not have a default user or the default user of the calling task's job was not a user object.
- E\$PARAM** The system call forced the Extended I/O System to attempt the physical attachment of the device referenced by the path\$name\$p parameter. The device had formerly been only logically attached. In the process, the Extended I/O System found that the logical attachment referred to a file driver (named, physical, or stream) that is not configured into your system. Hence the physical attachment is not possible.
- E\$PREPOSITION** One of the following situations occurred.
- The command line contained an invalid preposition value (a value greater than 3).
  - The command line contained a zero as the preposition value. This indicated that the same preposition was to be used as in the last C\$GET\$OUTPUT\$PATHNAME call. Unfortunately, C\$GET\$OUTPUT\$PATHNAME has not been called.
- E\$SHARE** While attempting to open a connection to the file, the Operating System detected another task writing to the file through another connection. However, when the connection was created, the owner specified that it could not be shared with writers. Therefore your task cannot access the file.
- E\$SPACE** One of the following situations occurred.
- While attempting to create a file or write into a file, the Operating System detected that there was no more space on the volume.
  - While attempting to create a file, the Operating System detected that the volume already contained the maximum number of files.

## C\$GET\$OUTPUT\$PATHNAME

C\$GET\$OUTPUT\$PATHNAME, a command parsing call, gets a pathname from the list of output pathnames in the parsing buffer.

```
preposition = RQ$C$GET$OUTPUT$PATHNAME(path$name$p, path$name$max,
                                         default$output$p, except$ptr);
```

## INPUT PARAMETERS

**path\$name\$max** A WORD that specifies the length in bytes of the string pointed to by the path\$name\$p parameter. The maximum length that you can specify is 256 bytes (255 characters for the pathname and one byte for the count).

**default\$output\$p** A POINTER to a STRING containing the command's default standard output. If the first invocation of this system call does not encounter a TO/OVER/AFTER preposition, the text of this parameter will be used as though it had appeared in the command line. The text must specify TO, OVER, or AFTER for the output mode. Examples: TO :CO: or TO :LP:.

## OUTPUT PARAMETERS

**preposition** A BYTE describing the preposition type that C\$GET\$OUTPUT\$PATHNAME encountered. You can pass this value to C\$GET\$OUTPUT\$CONNECTION when obtaining an output connection to the file. The value will be one of the following:

<u>Value</u>	<u>Meaning</u>
1	TO
2	OVER
3	AFTER

**path\$name\$p** A POINTER to a STRING that receives the next pathname in the pathname list.

**except\$ptr** A POINTER to a WORD in which the Human Interface returns a condition code.

# C\$GET\$OUTPUT\$PATHNAME

## DESCRIPTION

You should not call C\$GET\$OUTPUT\$PATHNAME before first calling C\$GET\$INPUT\$PATHNAME.

The first call to C\$GET\$OUTPUT\$PATHNAME retrieves the preposition (TO/OVER/AFTER) and the entire output pathname list; it then moves the parsing pointer to the next parameter. If the parsing buffer does not contain a preposition and pathname list, C\$GET\$OUTPUT\$PATHNAME uses the default pointed to by the default\$output\$p parameter (and does not move the parsing pointer). After retrieving the pathname list, C\$GET\$OUTPUT\$PATHNAME stores it in an internal buffer, returns the first pathname in the string pointed to by the path\$name\$p parameter, and returns the preposition in the preposition parameter. Succeeding calls to C\$GET\$OUTPUT\$PATHNAME return additional pathnames from the output pathname list (as well as the preposition), but they do not move the parsing pointer. C\$GET\$INPUT\$PATHNAME denotes the end of the pathname list by returning a zero-length string in the string pointed to by path\$name\$p.

C\$GET\$OUTPUT\$PATHNAME accepts wild-card characters in the last component of a pathname. It generates each output pathname based on this wild-carded pathname, the corresponding wild-carded pathname that was input to C\$GET\$INPUT\$PATHNAME, and the most recent input pathname returned by C\$GET\$INPUT\$PATHNAME.

The pathname returned by C\$GET\$OUTPUT\$PATHNAME can be used for any purpose. However, it is most often used in a call to C\$GET\$OUTPUT\$CONNECTION to obtain a connection to the file. In such a case, C\$GET\$OUTPUT\$CONNECTION processes the TO/OVER/AFTER preposition. If the pathname is used as input to a system call other than C\$GET\$OUTPUT\$CONNECTION, the interpretation of the TO/OVER/AFTER preposition is the user's responsibility.

## EXCEPTION CODES

E\$OK	No exceptional conditions were encountered.
E\$CONTEXT	The calling task's job was not created by the Human Interface.
E\$DEFAULT\$SO	The parsing buffer did not contain a preposition and output pathnames, so C\$GET\$OUTPUT\$PATHNAME attempted to use the default pointed to by default\$output\$p. However, this string contained an invalid preposition or pathname.

E\$LIMIT	At least one of the following situations occurred: <ul style="list-style-type: none"><li>• While creating an object, the Operating System detected a job's object limit having been exceeded. The job contained the task which invoked this system call.</li><li>• The calling task's job was not created by the Human Interface.</li></ul>
E\$MEM	The memory pool of the calling task's job does not currently have a block of memory large enough to allow this system call to run to completion.
E\$STRING	The pathname to be returned exceeds the length limit of 255 characters.
E\$STRING\$- BUFFER	The buffer pointed to by the path\$name\$p parameter was not large enough for the pathname to be returned.
E\$UNMATCHED\$- LISTS	The number of files in the input and output pathname is not the same.
E\$WILDCARD	The output pathname contains an invalid wild-card specification.

## C\$GET\$PARAMETER

GET\$PARAMETER, a command parsing call, gets a parameter from the parsing buffer.

```
more = RQ$C$GET$PARAMETER(name$p, name$max, value$p, value$max,
                           index$p, predict$list$p, except$ptr);
```

## INPUT PARAMETERS

name\$max	A WORD that specifies the length in bytes of the string pointed to by the name\$p parameter. The maximum length is 256 bytes (255 characters for the name and one byte for the count).
value\$max	A WORD that specifies the length in bytes of the string pointed to by the value\$p parameter. The maximum length is 65535 decimal bytes.
predict\$list\$p	A POINTER to a STRING\$TABLE, as described in Appendix C, that specifies the values that this system call accepts as prepositions. The predict\$list\$p POINTER should be zero if you do not intend to retrieve parameters that use prepositions.

## OUTPUT PARAMETERS

more	A BYTE value that indicates whether or not the current call to C\$GET\$PARAMETER returned a parameter. A value of 00h indicates that there are no more parameters (and that no parameter was returned); a value of 0FFh indicates that a parameter was returned.
name\$p	A POINTER to a STRING that receives the keyword portion of the parameter. If this parameter does not contain a keyword portion, the Human Interface returns a null (zero-length) string.
value\$p	A POINTER to a STRING\$TABLE, as described in Appendix C, that receives the value portion of the parameter. If the value portion contains a list of values separated by commas, the Human Interface returns the values to the string table one value per string.



`index$p` A POINTER to a BYTE that receives the index to the list of prepositions pointed to by `predict$list$p`. This index identifies the `name$p` keyword as a preposition and identifies it out of the list of possible prepositions. If the `predict$list$p` list is empty, or if the keyword name is not contained in the `predict$list$p` list, the system call returns a value of zero for the index. That is, the index will be non-zero only if a keyword exists and it is one of the prepositions in the `predict$list$p` list.

`except$ptr` A POINTER to a WORD in which the Human Interface returns a condition code.

## DESCRIPTION

C\$GET\$PARAMETER retrieves one parameter from the parsing buffer and moves the parsing pointer to the next parameter. The parameter can be one of the following:

- keyword/value-list parameter using parentheses
- keyword/value-list parameter using an equal sign
- keyword/value-list parameter with the keyword as a preposition
- value-list without a keyword

A description of the types, format, and syntax of acceptable parameters is provided in Chapter 2.

C\$GET\$PARAMETER places the keyword portion of the parameter in the string pointed to by `name$p`; it places the keyword list in the string table pointed to by `value$p`.

Without input from you, C\$GET\$PARAMETER cannot determine whether groups of characters separated by spaces are separate parameters or a single parameter that uses a preposition. C\$GET\$PARAMETER uses the list of prepositions that you supply in the string table pointed to by `predict$list$p` to determine the prepositions that can appear. When C\$GET\$PARAMETER retrieves a parameter, it obtains from the parsing buffer the next group of characters that are separated by spaces. Then it checks those characters against those in the `predict$list$p` list. If the characters match one of the values in the list, C\$GET\$PARAMETER realizes that the characters represent a preposition and not an entire parameter; it then obtains the next group of characters separated by spaces as the value portion of the parameter.

## EXCEPTION CODES

E\$OK No exceptional conditions were encountered.

- E\$CONTEXT                    The calling task's job was not an I/O job. Refer to the iRMX 86 EXTENDED I/O SYSTEM REFERENCE MANUAL for information about I/O jobs.
- E\$CONTINUED                The Operating System detected a continuation character in the parse buffer while performing the system call. This condition should only occur while parsing the contents of a buffer other than the command line buffer.
- E\$LIMIT                    At least one of the following situations occurred.

  - While creating an object, the Operating System detected that the object limit of the calling task's job had been reached.
  - The calling task's job was not an I/O job. Refer to the iRMX 86 EXTENDED I/O SYSTEM REFERENCE MANUAL for information about I/O jobs.
- E\$LIST                    At least one of the following situations occurred.

  - The parameter contains an unmatched parenthesis.
  - A value in the value list is missing or an improper value was entered. Examples of both these conditions follow:

<u>Value</u>	<u>Comments</u>
A,B,	No value following second comma.
A,B=C,D	The equal sign can not be used unless it is between quotes: 'B=C' is valid.
A,B(C,E),F	The parentheses can not be used in a value unless it is between quotes. However, A,B,(C,E),F is valid.
- E\$LITERAL                The Operating System detected a literal (quoted string) in the parsing buffer with no closing quote. This condition should only occur while parsing the contents of a buffer other than the command line buffer.
- E\$MEM                    The memory pool of the calling task's job does not currently have a block of memory large enough to allow this system call to run to completion.
- E\$PARAM                The predict\$list\$p parameter pointed to a string table, but the index\$p parameter was specified as zero.

- E\$PARSE\$TABLES** The Human Interface detected an error that should not occur unless someone inadvertently alters an internal table used by the Human Interface.
- E\$SEPARATOR** The Operating System detected an invalid command separator in the parsing buffer while performing this system call. This condition should only occur while parsing the contents of a buffer other than the command line buffer. The following is a list of invalid command separators: ><, <>, ||, |, [, and ].
- E\$STRING** One or more of the following conditions exist:
- The string to be returned as the parameter name exceeds the length limit of 255 characters.
  - One of the parameter values to be returned exceeds 255 characters in length.
- E\$STRING\$BUFFER** One or more of the following conditions exist:
- The string to be returned as the parameter name exceeds the buffer size provided by the user in the call.
  - The parameter values to be returned exceed the value-buffer size provided by the user in the call.

**C\$SEND\$COMMAND**

**C\$SEND\$COMMAND**, a command processing call, sends command lines to a command connection created by **C\$CREATE\$COMMAND\$CONNECTION** and, when the command is complete, invokes the command.

```
CALL RQ$C$SEND$COMMAND(command$conn, line$p, command$except$ptr,
                        except$ptr);
```

**INPUT PARAMETERS**

<b>command\$conn</b>	A TOKEN for the command connection that receives the command line.
<b>line\$p</b>	A POINTER to a STRING containing a command line to execute.

**OUTPUT PARAMETERS**

<b>command\$except\$ptr</b>	A POINTER to a WORD in which the Human Interface returns a condition code indicating the status of the invoked command. This parameter is undefined if an exceptional condition code is returned in the word pointed to by <b>except\$ptr</b> .
<b>except\$ptr</b>	A POINTER to a WORD in which the Human Interface returns a condition code indicating the status of the <b>C\$SEND\$COMMAND</b> system call.

**DESCRIPTION**

You can use this system call when you want to invoke a command programmatically instead of interactively. It stores a command line in the command connection created by the **C\$CREATE\$COMMAND\$CONNECTION** call, concatenates the command line with any others already stored there, and (if the command invocation is complete) invokes the command. The command can be any standard Human Interface command (as described in the **iRMX 86 OPERATOR'S MANUAL**) or a command that you create.

As described in greater detail in Chapter 3, a command invocation can contain several continuation marks. The continuation mark (&) indicates that the command line is continued on the next line. If the command line sent by **C\$SEND\$COMMAND** is continued on another line (that is, contains a continuation mark), the Human Interface returns an **E\$CONTINUED** exception code and does not invoke the command. You can then call **C\$SEND\$COMMAND** any number of times to send the continuation lines.

C\$SEND\$COMMAND concatenates the original command line and all continuation lines into a single command line in the command connection. It removes all continuation marks and all comments from this ultimate command line.

When the command invocation is complete (that is, the line sent by C\$SEND\$COMMAND does not contain a continuation mark) the Human Interface parses the command pathname from the command line. If no exception conditions halt the process at this point, the Human Interface requests the Application Loader to load and execute the command.

An Application Loader call creates an I/O job for the command. Then the Application Loader validates the header, group definition and segment definition records of the command's object file. Refer to the 8086 FAMILY UTILITIES USER'S GUIDE for explanations of segments, groups and object file formats.

C\$SEND\$COMMAND returns two condition codes: one for the C\$SEND\$COMMAND call and one for the invoked command. The word pointed to by the `except$ptr` parameter returns the C\$SEND\$COMMAND conditions, as described under the EXCEPTION CODES heading in this command description. The word pointed to by the `command$except$ptr` returns the invoked command's condition codes; the values returned depend on the command invoked. The `E$CONTROL$C` exception code can be returned at either place.

#### EXCEPTION CODES

E\$OK	No exceptional conditions were encountered.
E\$BAD\$GROUP	The object file represented by the command's pathname contained an invalid group definition record.
E\$BAD\$HEADER	The object file represented by the command's pathname does not begin with a header record for a loadable object module.
E\$BAD\$SEGMENT	The object file represented by the command's pathname contains an invalid segment definition record.
E\$CHECKSUM	At least one record of the object file represented by the command's pathname contains a checksum error. This situation could occur if the CHECKSUM amount calculated during the read operation did not match the CHECKSUM field of the record being read.
E\$CONTEXT	At least one of the following situations occurred. <ul style="list-style-type: none"> <li>● The calling task's job was not created by the Human Interface.</li> <li>● The device containing the object file was in the process of being detached.</li> </ul>

- The Extended I/O System was unable to attach the device containing the object file because the Basic I/O System has already attached the device.

## E\$CONTINUED

The Operating System detected a continuation character while scanning the command line pointed to by the line\$p parameter. This condition should occur if the command line is to continue on the next line.

## E\$DEVFD

The Extended I/O System attempted the physical attachment of a device that had formerly been only logically attached. In the process, the Extended I/O System found that the device and the device driver specified in the logical attachment were incompatible. The Operating System would not have returned this exception code if the device referred to by the line\$p parameter had been properly configured in the Extended I/O and/or Basic I/O subsystems.

## E\$EOF

The Application Loader encountered an unexpected end of file on the object file represented by the command's pathname.

## E\$EXIST

At least one of the following situations occurred.

- The Operating System detached the device containing the object file before it completed the loading operation.
- The token contained in the command\$conn parameter was not the token for a command connection.

## E\$FACCESS

The Operating System detected the user as not having READ access to the object file.

## E\$FIXUP

When the Application Loader loads an LTL (load-time-locatable) program, the Loader must adjust some of the addresses used in the code to reflect actual loaded code addresses. This adjustment is known as a fixup and is contained on a fixup record. The Application Loader detected an invalid fixup record in the object file. Refer to the iRMX 86 LOADER REFERENCE MANUAL for an explanation of LTL code.

## E\$FLUSHING

The Operating System detected that the device containing the object file was in the process of being detached.

E\$FNEXIST	<p>At least one of the following situations occurred.</p> <ul style="list-style-type: none"> <li>• A file in the command's pathname is either marked for deletion or does not exist. For example, the pathname A/B/C assumes that A and B are names for directories and C is the name of an object file. This exception code would be returned if A, B, or C was marked for deletion or did not exist.</li> <li>• While attaching the file specified in the line\$p parameter, the Extended I/O System attempted the physical attachment of the device as a named device. (This device had formerly been only logically attached.) It could not complete this process because the device's physical name does not exist.</li> </ul>
E\$FTYPE	<p>The command's pathname included the name of a data file as a directory. For example, the pathname A/B/C assumes that A and B are names for directories. This exception code would be returned if either A or B was something other than a directory.</p>
E\$ILLVOL	<p>The Extended I/O System attempted the physical attachment of a device that had formerly been only logically attached. During this process, the Extended I/O System examined the volume label and found that the volume does not contain named files. This prevented the Extended I/O System from completing physical attachment, because the named file driver was requested during logical attachment.</p>
E\$IO\$HARD	<p>While attempting to access the object file, the Operating System detected a hard I/O error.</p>
E\$IOMEM	<p>The Basic I/O System does not currently have a block of memory large enough to allow the Human Interface to create the connection necessary to allow this call to run to completion.</p>
E\$IO\$OPRINT	<p>While attempting to access the object file, the Operating System detected that the device was off-line. Operator intervention is required. C\$FORMAT\$EXCEPTION returns the value E\$IO\$NOT\$READY for this code.</p>
E\$IO\$SOFT	<p>While attempting to access the object file, the Operating System detected a soft I/O error. The error also occurred during retry.</p>
E\$IO\$UNCLASS	<p>An unknown type of I/O error occurred while trying to access the object file.</p>

E\$IO\$WRPROT           While attempting to obtain an input connection to the object file, the Operating System detected that the volume containing the file is write-protected.

E\$LIMIT                At least one of the following situations occurred.

- While creating the objects needed by this call, the Operating System detected the calling task's job already having reached the maximum number of objects that can exist simultaneously.
- The calling task's job , or the job's default user object, is currently involved in more than 255 (decimal) I/O operations.
- Either the newly created I/O job, or its default user, is currently involved in more than 255 (decimal) I/O operations.
- The calling task's job is not an I/O job. Refer to the iRMX 86 EXTENDED I/O SYSTEM REFERENCE MANUAL for information about I/O jobs.

E\$LITERAL             The Operating System detected a literal (quoted string) with no closing quote while scanning the contents of the command line pointed to by the line\$p parameter.

E\$LOG\$NAME\$-  
NEXIST                 The command's pathname contains an explicit logical name but the Extended I/O System was unable to find this name in the object directories of the local job, the global job, and the root job.

E\$LOG\$NAME\$-  
SYNTAX                The command's pathname contains a logical name. However, the logical name contains unmatched colons, is longer than 12 characters, or contains invalid characters.

E\$MEDIA                The device containing the object file was off-line.

E\$MEM                  At least one of the following situations occurred.

- The memory pool of the calling task's job does not currently have a block of memory large enough to allow this system call to run to completion.
- The memory pool of the newly-created I/O job does not currently have a block of memory large enough to allow the initial task to start running.
- The memory pool of the Basic I/O System job does not currently have a block of memory large enough to allow the Application Loader to run.



- E\$NO\$LOADER\$MEM** At least one of the following situations occurred.
- The memory pool of the newly-created I/O job does not currently have a block of memory large enough to allow the Loader to run.
  - The memory pool of the Basic I/O System's job does not currently have a block of memory large enough to allow the Application Loader to run.
- E\$NO\$MEM** The Application Loader attempted to load PIC or LTL groups or segments. However, the memory pool of the newly-created I/O job does not currently contain a block of memory large enough to accommodate these groups or segments. Refer to the IRMX 86 LOADER REFERENCE MANUAL for an explanation of loading PIC or LTL groups or segments.
- E\$NO\$PREFIX** The command's pathname contained no explicit prefix (no logical name), so the Extended I/O System attempted to use the default prefix. However, the default prefix is either undefined, or it is not a valid device connection or file connection.
- E\$NO\$START** The object file represented by the command-pathname does not specify the entry point for the program being loaded.
- E\$NOT\$CONNECTION** The token specified in either the default\$sci or default\$co parameter of the system call C\$CREATE\$COMMAND\$CONNECTION no longer refers to a valid connection.
- E\$NOT\$LOG\$NAME** The Operating System detected that the command pathname contains a logical name. The logical name refers to an object that is neither a device connection nor a file connection.
- E\$NO\$USER** The calling task's job does not have a valid default user.
- E\$PARAM** The Extended I/O System attempted the physical attachment of a device containing the object file. This device had formerly been only logically attached. While attempting this, the Extended I/O System found that the logical attachment referred to a file driver (named, physical, or stream) that is not configured into your system. Hence the physical attachment is not possible.

E\$PARSE\$TABLES	The Human Interface has detected an error that should not occur unless someone alters an internal table used by this subsystem.
E\$REC\$FORMAT	At least one record in the object file contains a record format error.
E\$REC\$LENGTH	The object file contains a record that is longer than the Loader's maximum record length. The Loader's maximum record length is a parameter specified during the configuration of the Loader. Refer to the iRMX 86 CONFIGURATION GUIDE for details.
E\$REC\$TYPE	<p>The Application Loader detected one of the following situations while attempting to load the object file.</p> <ul style="list-style-type: none"> <li>● At least one record in the file being loaded is of a type that the Loader cannot process.</li> <li>● The Loader has encountered records in a sequence that it cannot process.</li> </ul>
E\$SEG\$BOUNDS	The Application Loader created multiple segments in which to load information. One of the data records in the object file specified a load address outside of the created segments.
E\$SEPARATOR	The Operating System detected an invalid separator while scanning the command line. The following is a list of the invalid command separators: ><, <>,   ,  , [, and ].
E\$STRING	The size of the command's pathname exceeds the length limit of 255 (decimal) characters.
E\$STRING\$BUFFER	The size of the command's pathname exceeds the size of the command name buffer specified during the configuration of the Human Interface.
E\$TIME	The Extended I/O System was unable to start the command because the calling task's job was not created by the Human Interface subsystem.
E\$TYPE	The command\$conn parameter contains a token for an object that is not a command connection.

## C\$SEND\$CO\$RESPONSE

C\$SEND\$CO\$RESPONSE, a message processing call, sends a message to :CO: and reads a response from :CI:.

```
CALL RQ$C$SEND$CO$RESPONSE(response$p, response$max, message$p,
                             except$ptr);
```

## INPUT PARAMETERS

**message\$p** A POINTER to a STRING containing the message to be sent to :CO:. If zero, no message is sent.

**response\$max** A WORD that specifies the length in bytes of the string pointed to by the response\$p parameter. If response\$max is zero, no response from :CI: will be requested; control returns to the calling task immediately.

## OUTPUT PARAMETERS

**response\$p** A POINTER to a STRING that receives the operator's response from :CI:.

**except\$ptr** A POINTER to a WORD in which the Human Interface returns a condition code.

## DESCRIPTION

When used with all its features, C\$SEND\$CO\$RESPONSE sends the string pointed to by message\$p to :CO: and waits for a response from :CI:. It places this response in the string pointed to by response\$p. However, if message\$p is zero, C\$SEND\$CO\$RESPONSE omits sending the message to :CO:; if either response\$max or response\$p is zero, it does not wait for a response from :CI:. Therefore, the operations performed by C\$SEND\$CO\$RESPONSE depend on the values of the message\$p and response\$max parameters, as follows:

<u>message\$p</u>	<u>response\$max</u>	<u>Action</u>
zero	zero	Perform no I/O
zero	non-zero	Send no message, wait for input
non-zero	non-zero	Send message, wait for input
non-zero	zero	Send message, don't wait

If C\$SEND\$CO\$RESPONSE requests a response from :CI:, no other output can be displayed at :CO: until C\$SEND\$CO\$RESPONSE receives a line terminator from :CI:.. However, the operator can choose to ignore the displayed message by entering a line terminator only.

The main distinction between C\$SEND\$CO\$RESPONSE and C\$SEND\$EO\$RESPONSE calls is that C\$SEND\$EO\$RESPONSE always sends messages to and receives messages from the operator's terminal; input and output cannot be redirected to another device. In contrast, C\$SEND\$CO\$RESPONSE sends messages to :CO: and receives messages from :CI:; therefore, programs such as SUBMIT can redirect this input and output.

#### EXCEPTION CODES

E\$OK	No exceptional conditions were encountered.
E\$CONTEXT	At least one of the following situations occurred. <ul style="list-style-type: none"><li>• The Operating System detected a zero value for the object directory size. This indicates that the calling task's job was not created by the Human Interface.</li><li>• The connection to :CI: was not open for reading or the connection to :CO: was not open for writing.</li><li>• The connection to :CI: or :CO: was not open.</li><li>• The Operating System detected that the connections to :CI: and :CO: were opened with A\$OPEN rather than S\$OPEN.</li><li>• When attempting to read or write to :CI: or :CO:, the Extended I/O System issued an invalid stream file request.</li></ul>
E\$EXIST	The file representing :CI: or :CO: is marked for deletion or does not exist.
E\$FLUSHING	The Operating System detected that the device containing the :CI: and :CO: files was in the process of being detached.
E\$IO\$HARD	While attempting to access the :CI: or :CO: file, the Operating System detected a hard I/O error.
E\$IO\$OPRINT	While attempting to access the :CI: or :CO: file, the Operating System detected that the device was off-line. Operator intervention is required. C\$FORMAT\$EXCEPTION returns the value E\$IO\$NOT\$READY for this code.

E\$IO\$SOFT            While attempting to access the :CI: or :CO: file, the Operating System detected a soft I/O error. The error also occurred during retry.

E\$IO\$UNCLASS        An unknown type of I/O error occurred while trying to access the :CI: or :CO: file.

E\$IO\$WRPROT        While attempting to obtain a connection to the :CO: file, the Operating System detected that the volume containing the file is write-protected.

E\$LIMIT            At least one of the following situations occurred.

- The calling task's job contains the maximum number of objects that can exist simultaneously.
- The calling task's job, or the job's default user object, is currently involved in more than 255 (decimal) I/O operations.
- The calling task's job was not created by the Human Interface.

E\$MEM              The memory pool of the calling task's job does not currently have block of memory large enough to allow this system call to run to completion.

E\$NOT\$CONNECTION   Although the Operating System obtained a token for a connection to :CI: and :CO:, at least one of these tokens does not represent a valid connection.

E\$PARAM            The Operating System detected an attempt to write beyond the end of a physical file.

E\$SPACE            One of the following situations occurred.

- While processing a write call, the Operating System detected a full volume. Therefore, the Extended I/O System was unable to complete the requested writing operation.
- The Operating System detected an attempt to write beyond the end of a physical file.

E\$SUPPORT          The connection to :CI: or :CO: was created by a task that is not contained in the calling task's job.

E\$TIME             The calling task's job was not created by the Human Interface.

**C\$SEND\$EO\$RESPONSE**

**C\$SEND\$EO\$RESPONSE**, a message processing call, sends a message to and reads a response from the operator's terminal.

CALL RQ\$C\$SEND\$EO\$RESPONSE(response\$p, response\$max, message\$p,  
except\$ptr);

**INPUT PARAMETERS**

message\$p	A POINTER to a STRING containing the message to be sent to the operator's terminal. If zero, no message is sent.
response\$max	A WORD that specifies the length in bytes of the string pointed to by the response\$p parameter. If response\$max is zero, no response from the operator's terminal will be requested; control returns to the calling task immediately.

**OUTPUT PARAMETERS**

response\$p	A POINTER to a STRING that receives the operator's response from the terminal.
except\$ptr	A POINTER to a WORD in which the Human Interface returns a condition code.

**DESCRIPTION**

When used with all its features, **C\$SEND\$EO\$RESPONSE** sends the string pointed to by message\$p to the operator's terminal and waits for a response from the operator. It places this response in the string pointed to by response\$p. However, if message\$p is zero, **C\$SEND\$EO\$RESPONSE** omits sending the message to the operator; if either response\$max or response\$p is zero, it does not wait for a response. Therefore, the operations performed by **C\$SEND\$EO\$RESPONSE** depend on the values of the message\$p and response\$max parameters, as follows:

<u>message\$p</u>	<u>response\$max</u>	<u>Action</u>
zero	zero	Perform no I/O
zero	non-zero	Send no message, wait for input
non-zero	non-zero	Send message, wait for input
non-zero	zero	Send message, don't wait

If C\$SEND\$EO\$RESPONSE requests a response from the terminal, no other output can be displayed at the terminal until C\$SEND\$EO\$RESPONSE receives a line terminator from the operator. However, the operator can choose to ignore the displayed message by entering a line terminator only.

The main distinction between C\$SEND\$CO\$RESPONSE and C\$SEND\$EO\$RESPONSE calls is that C\$SEND\$EO\$RESPONSE always sends messages to and receives messages from the operator's terminal; input and output cannot be redirected to another device. In contrast, C\$SEND\$CO\$RESPONSE sends messages to :CO: and receives messages from :CI:; therefore programs such as SUBMIT can redirect this input and output.

#### EXCEPTION CODES

E\$OK	No exceptional conditions were encountered.
E\$CONTEXT	At least one of the following situations occurred. <ul style="list-style-type: none"> <li>● The Operating System detected a zero value for the object directory size. This indicates that the calling task's job was not created by the Human Interface.</li> <li>● The connection to the terminal was not open for reading or for both reading and writing.</li> <li>● The connection to the terminal was closed.</li> <li>● The Operating System detected that the connection to the terminal was opened with A\$OPEN rather than S\$OPEN.</li> <li>● When attempting to read or write to the terminal, the Extended I/O System issued an invalid stream file request.</li> </ul>
E\$EXIST	The connection to the terminal is marked for deletion or does not exist.
E\$FLUSHING	The Operating System detected that the terminal was in the process of being detached.
E\$I/O\$OPRINT	While attempting to access the terminal, the Operating System detected that the device was off-line. Operator intervention is required. C\$FORMAT\$EXCEPTION returns the value E\$I/O\$NOT\$READY for this code.

- E\$LIMIT At least one of the following situations occurred.
- The calling task's job contains the maximum number of objects that can exist simultaneously.
  - The calling task's job, or the job's default user object, is currently involved in more than 255 (decimal) I/O operations.
  - The calling task's job was not created by the Human Interface.
- E\$MEM The memory pool of the calling task's job does not currently have block of memory large enough to allow this system call to run to completion.
- E\$NOT\$CONNECTION Although the Operating System obtained a token for a connection to the terminal the token does not represent a valid connection.
- E\$PARAM The Operating System detected an attempt to write beyond the end of a physical file.
- E\$SUPPORT The connection to the terminal was created by a task that is not contained in the calling task's job.
- E\$TIME The calling task's job was not created by the Human Interface.



## C\$SET\$PARSE\$BUFFER

C\$SET\$PARSE\$BUFFER, a command parsing call, permits parsing the contents of a buffer other than the command line buffer whenever the parsing system calls are used.

```
offset = RQ$C$SET$PARSE$BUFFER(buff$p, buff$max, except$ptr);
```

## INPUT PARAMETERS

**buff\$p** A POINTER to a buffer containing the text to be parsed. If the buff\$p is zero, the buffer used for parsing reverts to the command line buffer and the buff\$max parameter is ignored.

**buff\$max** A WORD that specifies the length in bytes of the string pointed to by the buff\$p parameter.

## OUTPUT PARAMETERS

**offset** A WORD in which the Human Interface places the byte offset from the start of the parsing buffer of the last byte parsed in the previous parsing buffer.

**except\$ptr** A POINTER to a WORD in which the Human Interface returns a condition code.

## DESCRIPTION

C\$SET\$PARSE\$BUFFER allows you to parse buffers other than the command line. You can change buffers at will; you can also revert to the command line parsing buffer by calling C\$SET\$PARSE\$BUFFER with buff\$p=0. However, only one parsing buffer per job can be active at any given time.

When called, C\$SET\$PARSE\$BUFFER sets the parsing pointer to the beginning of the specified buffer. However, it also returns a value (in the offset parameter) that identifies the last byte parsed in the previous parsing buffer. This gives you the ability, when switching back to the previous buffer, of positioning the parsing pointer to its previous position with successive calls to C\$GET\$CHAR.

Note that C\$SET\$PARSE\$BUFFER does not affect the buffer from which C\$GET\$INPUT\$PATHNAME and C\$GET\$OUTPUT\$PATHNAME retrieve pathnames. These system calls always obtain their pathnames from the command line.

EXCEPTION CODES

E\$OK	No exceptional conditions were encountered.
E\$CONTEXT	The calling task's job is not an I/O job. Refer to the IRMX 86 EXTENDED I/O SYSTEM REFERENCE MANUAL for information about I/O jobs.
E\$LIMIT	At least one of the following situations occurred. <ul style="list-style-type: none"><li>● The calling task's job contains the maximum number of objects that can exist simultaneously.</li><li>● The Operating System detected an object directory that has already reached the maximum object directory size. This indicates that the calling task's job was not created by the Human Interface.</li></ul>
E\$MEM	The memory pool of the calling task's job does not currently have block of memory large enough to allow this system call to run to completion.

\*\*\*

## CHAPTER 9. CONFIGURATION OF THE HUMAN INTERFACE

The Human Interface is a configurable part of the Operating System. It contains several options that you can adjust to meet your specific needs. To help you make configuration choices, Intel provides three kinds of information:

- A list of configurable options
- Detailed information about the options
- Procedures to allow you to specify your choices

The balance of this chapter provides the first category of information. To obtain the second and third categories of information, refer to the *IRMX 86 CONFIGURATION GUIDE*.

Human Interface configuration consists of two parts: resident configuration and nonresident configuration. Resident configuration involves configuring the portion of the Human Interface that resides in system memory at all times. This configuration takes place during the configuration of the entire Operating System, when you adjust parameters, include or exclude layers of the Operating System, and generate an executable version of the Operating System. You cannot change the resident configuration without reconfiguring the entire Operating System. Nonresident configuration involves setting up an iRMX 86 directory structure and placing information about users into iRMX 86 files. The nonresident configuration information must be present when the application system starts running, but you can modify the information in the nonresident configuration files while the system is running. For the new nonresident configuration to take effect, you must reinitialize your application system.

### RESIDENT CONFIGURATION

When you perform the resident Human Interface configuration, you can modify parameters of the Human Interface that affect all Human Interface users. These include:

- Information about the Human Interface's initial job, such as minimum and maximum memory pool size and whether jobs created by the Human Interface expect to use the 8087 Numeric Processor Extension.
- Information about the initial user (or single user, if a single-access system), including terminal name, user ID, maximum priority, pathname of initial program, and default directory.

## CONFIGURATION OF THE HUMAN INTERFACE

- Information about the jobs created by the Human Interface, including minimum and maximum memory pool sizes.
- Initial size of the buffer that the Human Interface uses when constructing commands.
- Maximum length of a command pathname.
- List of directories that the Human Interface automatically searches, in order, when trying to find a command.
- Pathname of the directory assigned to the logical name :SYSTEM: and a list of pathnames and the logical names that you want the Human Interface to assign upon initialization.
- Whether the Human Interface includes an initial program that is linked to the Human Interface and used for all operators (resident initial program), or whether a separate initial program is used for each operator. If you include a resident initial program, you can also specify its pathname.

### NONRESIDENT CONFIGURATION

The nonresident configuration involves specifying information about the terminals and users that access a multi-access Human Interface.

For each terminal in the system you can specify:

- Terminal name
- Associated user name
- Memory partition size
- Maximum priority
- Pathname of the initial program

For each user in the system you can specify

- User ID
- Password
- Memory partition size
- Default prefix
- Pathname of the initial program
- Maximum job priority

\*\*\*

APPENDIX A. HUMAN INTERFACE TYPE DEFINITIONS

The type definitions used in Human Interface system call description are defined in Table A-1.

Table A-1. Type Definitions

Type	Definition
BYTE	An unsigned, eight-bit, binary number.
WORD	An unsigned, two-byte, binary number.
INTEGER	A signed, two-byte, binary number that is stored in two's complement form.
POINTER	Two consecutive words containing the base of a segment and the offset into that segment. The offset must be in the word having the lower address.
SELECTOR	A 16-bit quantity that is equivalent to the base portion of a pointer. Your PL/M compiler may not support this data type.
TOKEN	A word or selector whose value identifies an object. A TOKEN can be declared literally a WORD or a SELECTOR, depending on your needs.
STRING	A sequence of consecutive bytes. The value contained in the first byte is the number of bytes in the rest of the string. Since a string contains only a single byte in which to store the count, the maximum number of characters that a string can contain is 255. A zero count specifies a null string.
STRING\$TABLE	A count byte followed by a sequence of consecutive strings. The value contained in the count byte is the number of strings in the rest of the string table. Since the string table contains only a single byte in which to store the count, the maximum number of strings that a string table can contain is 255. A zero count specifies a null string table.

\*\*\*



## APPENDIX B. HUMAN INTERFACE EXCEPTION CODES

Like other iRMX 86 software systems, the Human Interface returns a condition code whenever a Human Interface call is invoked. If the call executes without error, the Human Interface returns the code E\$OK. When an error is encountered during call execution, an exceptional condition code is returned. The exceptional condition code may be returned either from the Human Interface or from one of the other iRMX 86 layers residing below it.

The exception codes listed in Table B-1 are unique to the Human Interface.

Table B-1. Human Interface Exception Codes

Programmer Errors:	
E\$PARSE\$TABLES	8080h
E\$JOB\$TABLES	8081h
E\$DEFAULT\$SO	8083h
E\$STRING	8084h
Environmental Errors:	
E\$OK	0000H
E\$LITERAL	0080H
E\$STRING\$BUFFER	0081H
E\$SEPARATOR	0082H
E\$CONTINUED	0083H
E\$INVALID\$NUMERIC	0084H
E\$LIST	0085H
E\$WILDCARD	0086H
E\$PREPOSITION	0087H
E\$PATH	0088H
E\$CONTROL\$C	0089H
E\$CONTROL	008AH
E\$UNMATCHED\$LISTS	008BH

Other exception codes may be issued during Human Interface operations. The hexadecimal values of these exception conditions fall into ranges based on the iRMX 86 layer that first detects the condition. Table B-2 lists the layers and their respective ranges.

HUMAN INTERFACE EXCEPTION CODES

Table B-2. Exception Code Ranges

System	Environmental	Programming
Nucleus	0 to 1FH	8000 to 801FH
Basic I/O System	20 to 3FH	8020 to 803FH
Extended I/O System	40 to 5FH	8040 to 805FH
Application Loader	60 to 7FH	8060 to 807FH
Human Interface	80 to AFH	8080 to 80AFH
Universal Development Interface	C0 to DFH	80C0 to 80DFH
Reserved *	130 to 14FH	8130 to 814FH
<p>* Exception codes in this range could occur if you are a user of an iRMX system with iMMX 800 software. Refer to iMMX 800 MULTIBUS MESSAGE EXCHANGE REFERENCE MANUAL for an explanation of exception conditions within this range.</p>		

Table B-2 provides a minimum of information about an exception condition. In most cases, the exception condition must be considered in terms of the unique circumstances that caused the condition. Table B-3 is provided to guide you to the most appropriate manual. The appropriate iRMX 86 manuals have more detailed descriptions of the meanings. The appropriate manual is listed in the column marked "Manuals".



HUMAN INTERFACE EXCEPTION CODES

Table B-3. iRMX™ 86 Condition Codes

Hex. Value	Mnemonic	Manuals N B E L H	Meaning
0H	E\$OK	* * * * *	No exceptional conditions (normal)
Environmental Conditions			
1H	E\$TIME	* * * * *	A time limit (possibly a limit of zero time) expired without a task's request being satisfied.
2H	E\$MEM	* * * * *	Insufficient available memory to satisfy a task's request.
3H	E\$BUSY	*	Another task currently has access to data protected by a region.
4H	E\$LIMIT	* * * * *	A task attempted an operation which, if it had been successful, would have violated a Nucleus-enforced limit.
5H	E\$CONTEXT	* * * * *	A system call was issued out of proper context.
6H	E\$EXIST	* * * * *	A token parameter has a value which is not the token of an existing object.
7H	E\$STATE	*	A task attempted an operation which would have caused an impossible transition of a task's state.
8H	E\$NOT\$CON- FIGURED	* * * * *	This system call is not part of the present configuration.
9H	E\$INTERRUPT\$- SATURATION	*	An interrupt task has accumulated the maximum allowable amount of SIGNAL\$INTERRUPT requests.
0AH	E\$INTERRUPT\$- OVERFLOW	*	An interrupt task has accumulated more than the maximum allowable amount of SIGNAL\$INTERRUPT requests.
N Nucleus Reference Manual		L Loader Reference Manual	
B Basic I/O System Ref Manual		H Human Interface Reference Manual	
E Extended I/O Sys Ref Manual			

HUMAN INTERFACE EXCEPTION CODES

Table B-3. iRMX™ 86 Condition Codes (continued)

Hex. Value	Mnemonic	Manuals N B E L H	Meaning
Environmental Conditions (continued)			
20H	E\$FEXIST	* *	File already exists.
21H	E\$FNEXIST	* * * *	File does not exist.
22H	E\$DEVFD	* * *	Device and file driver are incompatible.
23H	E\$SUPPORT	* * * *	Combination of parameters not supported.
24H	E\$EMPTY\$- ENTRY	* *	The specified slot in a directory file is empty.
25H	E\$DIR\$END	* *	The specified slot is beyond the end of a directory file.
26H	E\$FACCESS	* * * *	File access not granted.
27H	E\$FTYPE	* * *	Incompatible file type.
28H	E\$SHARE	* * * *	Improper file sharing requested.
29H	E\$SPACE	* *	No space left.
2AH	E\$IDDR	* *	Invalid device driver request.
2BH	E\$IO	* * * *	An I/O error occurred.
2CH	E\$FLUSHING	* * * *	Connection specified in call was deleted before the operation was completed.
2DH	E\$ILLVOL	* * *	Invalid volume name.
2EH	E\$DEV\$OFF- LINE	*	The device being accessed is now offline.
2FH	E\$IFDR	* *	Invalid file driver request.
N Nucleus Reference Manual B Basic I/O System Ref Manual E Extended I/O Sys Ref Manual		L Loader Reference Manual H Human Interface Reference Manual	

HUMAN INTERFACE EXCEPTION CODES

Table B-3. iRMX™ 86 Condition Codes (continued)

Hex. Value	Mnemonic	Manuals N B E L H	Meaning
Environmental Conditions (continued)			
40H	E\$LOG\$NAME\$- SYNTAX	* *	The specified pathname contains a logical name, but the logical name has an invalid syntax.
41H	E\$CANNOT\$- CLOSE	*	The Extended I/O System was not able to transfer remaining data in buffers to output device.
42H	E\$IOMEM	* *	The Basic I/O System has insufficient memory to process a request.
44H	E\$MEDIA	* *	The device containing a specified file is not online.
45H	E\$LOG\$NAME\$- NEXIST	* *	The Extended I/O System was unable to find a specified logical name in the object directories that it checks.
46H	E\$NOT\$OWNER	*	The user who attempted to detach the device is not the owner of the device.
47H	E\$IO\$JOB	*	The Extended I/O System cannot create an I/O job because the size specified for the object directory is too small.
50H	E\$IO\$UNCLASS	*	An unknown type of I/O error occurred.
51H	E\$IO\$SOFT	* *	A soft I/O error occurred. A retry might be successful.
52H	E\$IO\$HARD	* *	A hard I/O error occurred. A retry is probably useless.
53H	E\$IO\$OPRINT	* *	The device was off-line. Operator intervention is required. C\$FORMAT\$EXCEPTION returns the value E\$IO\$NOT\$READY for this code.
54H	E\$IO\$WRPROT	* *	The volume is write-protected.
N Nucleus Reference Manual		L Loader Reference Manual	
B Basic I/O System Ref Manual		H Human Interface Reference Manual	
E Extended I/O Sys Ref Manual			

HUMAN INTERFACE EXCEPTION CODES

Table B-3. iRMX™ 86 Condition Codes (continued)

Hex. Value	Mnemonic	Manuals N B E L H	Meaning
Environmental Conditions (continued)			
60H	E\$AB\$ADDRESS	*	An absolute object program was loaded into system protected memory area.
61H	E\$BAD\$GROUP	* *	Invalid group component in the a group definition record.
62H	E\$BAD\$-HEADER	* *	Invalid header record in the object file.
63H	E\$BAD\$SEG-DEF	* *	Invalid segment definition record.
64H	E\$CHECKSUM	* *	A checksum error occurred while reading an object record.
65H	E\$EOF	* *	Unexpected end of file encountered while reading object records.
66H	E\$FIXUP	* *	Invalid fixup record in the object file.
67H	E\$NO\$LOADER\$MEM	* *	Insufficient memory to satisfy loader dynamic memory requirements.
68H	E\$NO\$MEM	* *	Insufficient memory to create PIC/LTL segments.
69H	E\$REC\$FORMAT	* *	Invalid record format encountered.
6AH	E\$REC\$-LENGTH	* *	Record length of an object record exceeds configured loader-buffer size.
6BH	E\$REC\$TYPE	* *	Invalid record type encountered in the object file.
6CH	E\$NO\$START	* *	Start address not found.
N Nucleus Reference Manual B Basic I/O System Ref Manual E Extended I/O Sys Ref Manual		L Loader Reference Manual H Human Interface Reference Manual	

HUMAN INTERFACE EXCEPTION CODES

Table B-3. iRMX™ 86 Condition Codes (continued)

Hex. Value	Mnemonic	Manuals N B E L H	Meaning		
Environmental Conditions (continued)					
6DH	E\$JOB\$SIZE	* *	Maximum job-size specified is less than the memory requirement specified in the object file.		
6EH	E\$OVERLAY	*	Overlay name does not match with any of the overlay module names.		
6FH	E\$LOADER \$SUPPORT	* *	The object file being loaded requires features not supported by the configured loader.		
70H	E\$SEG\$ BOUNDS	*	One of the data records in a module loaded by the Application Loader referred to an address outside the segment created for it.		
80H	E\$LITERAL	*	The parse buffer contains a literal with no closing quote.		
81H	E\$STRING\$- BUFFER	*	The string to be returned as the parameter name exceeds the size of the buffer the user provided in the call.		
82H	E\$SEPARA- TOR	*	The parse buffer contains a command separator.		
83H	E\$CONTINUED	*	The parse buffer contains a continuation character.		
84H	E\$INVALID\$- NUMERIC	*	A numeric value contains invalid characters.		
85H	E\$LIST	*	The last value of the value list is missing.		
86H	E\$WILDCARD	*	A wild-card character appears in an invalid context, such as an intermediate component of a pathname.		
<table style="width: 100%; border: none;"> <tr> <td style="width: 50%; border: none;">                     N Nucleus Reference Manual                      B Basic I/O System Ref Manual                      E Extended I/O Sys Ref Manual                 </td> <td style="width: 50%; border: none;">                     L Loader Reference Manual                      H Human Interface Reference Manual                 </td> </tr> </table>				N Nucleus Reference Manual B Basic I/O System Ref Manual E Extended I/O Sys Ref Manual	L Loader Reference Manual H Human Interface Reference Manual
N Nucleus Reference Manual B Basic I/O System Ref Manual E Extended I/O Sys Ref Manual	L Loader Reference Manual H Human Interface Reference Manual				

HUMAN INTERFACE EXCEPTION CODES

Table B-3. iRMX™ 86 Condition Codes (continued)

Hex. Value	Mnemonic	Manuals N B E L H	Meaning
Environmental Conditions (continued)			
87H	E\$PREPOSITION	*	The same preposition as on the the command line was indicated, but can not be used.
88H	E\$PATH	*	The command line specifies an invalid pathname.
89H	E\$CONTROL\$C	*	The user typed CONTROL-C while the command was being loaded.
8AH	E\$CONTROL	*	The command line contains an invalid control.
8BH	E\$UNMATCHED\$LISTS	*	There were no more input pathnames although the output pathname list was not empty.
Programmer Errors			
8000H	E\$ZERO\$-DIVIDE	*	A task attempted to divide by zero.
8001H	E\$OVERFLOW	*	An overflow interrupt occurred.
8002H	E\$TYPE	* * * * *	A token parameter referred to an existing object that is not of the required type.
8003H	E\$BOUNDS	*	A task attempted to access beyond the end of a segment.
8004H	E\$PARAM	* * * * *	A parameter which is neither a token nor an offset has an invalid value.
N Nucleus Reference Manual B Basic I/O System Ref Manual E Extended I/O Sys Ref Manual		L Loader Reference Manual H Human Interface Reference Manual	

HUMAN INTERFACE EXCEPTION CODES

Table B-3. iRMX™ 86 Condition Codes (continued)

Hex. Value	Mnemonic	Manuals N B E L H	Meaning
Programmer Errors (continued)			
8005H	E\$BAD\$CALL	* *	The I/O System code has been damaged, probably due to a bug in an application task. Recovery is not possible.
8006H	E\$ARRAY\$- BOUNDS	*	Hardware or software has detected an array overflow.
8007H	E\$NDP\$- STATUS	*	An 8087 Numeric Processor Extension error has been detected; Operating System extensions can return the status of the 8087 to the exception handler.
8008H	E\$CHECK\$EX- CEPTION	*	A software interrupt 17 has occurred.
8021H	E\$NOUSER	* * *	No default user.
8022H	E\$NOPREFIX	* * *	No default prefix.
8040H	E\$NOT\$LOG\$- NAME	* *	Specified object is not a device connection or file connection.
8041H	E\$NOT\$- DEVICE	*	A token parameter referred to an existing object that is not, but should be, a device connection.
8042H	E\$NOT\$CON- NECTION	*	A token parameter referred to an existing object that is not, but should be, a file connection.
8060H	E\$JOB\$PARAM	* *	The maximum job-size specified is less than the minimum job-size.
8080H	E\$PARSE\$- TABLES	*	There is an error in the internal parse tables.
N Nucleus Reference Manual B Basic I/O System Ref Manual E Extended I/O Sys Ref Manual		L Loader Reference Manual H Human Interface Reference Manual	

HUMAN INTERFACE EXCEPTION CODES

Table B-3. iRMX™ 86 Condition Codes (continued)

Hex. Value	Mnemonic	Manuals N B E L H	Meaning						
Programmer Errors (continued)									
8081H	E\$JOB\$- TABLES	*	An internal Human Interface table was overwritten, causing it to contain an invalid value.						
8083H	E\$DEFAULT\$SO	*	The default output name STRING is invalid.						
8084H	E\$STRING	*	The pathname to be returned exceeds 255 characters in length.						
<table style="width: 100%; border: none;"> <tr> <td style="width: 50%; border: none;">N Nucleus Reference Manual</td> <td style="width: 50%; border: none;">L Loader Reference Manual</td> </tr> <tr> <td style="border: none;">B Basic I/O System Ref Manual</td> <td style="border: none;">H Human Interface Reference Manual</td> </tr> <tr> <td style="border: none;">E Extended I/O Sys Ref Manual</td> <td style="border: none;"></td> </tr> </table>				N Nucleus Reference Manual	L Loader Reference Manual	B Basic I/O System Ref Manual	H Human Interface Reference Manual	E Extended I/O Sys Ref Manual	
N Nucleus Reference Manual	L Loader Reference Manual								
B Basic I/O System Ref Manual	H Human Interface Reference Manual								
E Extended I/O Sys Ref Manual									

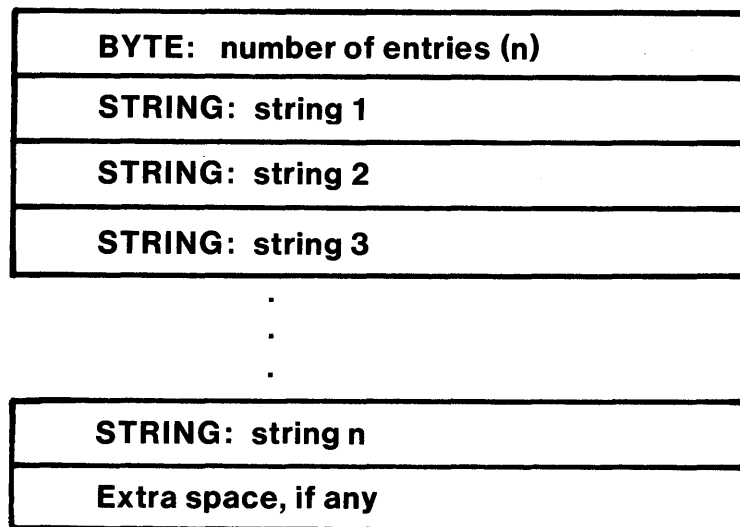
\*\*\*



## APPENDIX C. STRING TABLE FORMAT

The iRMX 86 Operating System uses structures called strings to store groups of ASCII characters (such as pathnames). The Operating System assumes a string to be a series of consecutive bytes broken into two portions: a count byte and text bytes. The first byte in the string is the count byte; its value is set to the number of bytes in text portion of the string. The text bytes contain the substance of the string.

The Operating System also uses another structure called a string table. A string table consists of a count byte and a series of consecutive strings. As with the string, the first byte in the string table is the count byte; its value is set to the number of strings in the string table. The diagram in Figure C-1 shows the string\$table parameter format.



1119

Figure C-1. String Table Format

---

## STRING TABLE FORMAT

### EXAMPLE:

Assume you wish to generate a string table containing the words HAPPY, GLAD, and SAD. The following declarations would be needed:

```
DECLARE
    p$table(*) BYTE DATA(3,          /* NUMBER OF STRINGS */
        5, 'HAPPY',
        4, 'GLAD',
        3, 'SAD' );
```

\*\*\*

## INDEX

Underscored entries are primary references.

AFTER preposition 3-2  
ampersand (&) 3-3

Basic I/O System 2-1  
BYTE A-1

C\$CREATE\$COMMAND\$CONNECTION system call 2-3, 5-1, 8-4  
C\$DELETE\$COMMAND\$CONNECTION system call 2-3, 5-3, 8-8  
C\$FORMAT\$EXCEPTION system call 4-4, 8-9  
C\$GET\$CHAR system call 3-15, 3-17, 8-11  
C\$GET\$COMMAND\$NAME system call 3-17, 8-13  
C\$GET\$INPUT\$CONNECTION system call 3-6, 4-1, 8-15  
C\$GET\$INPUT\$PATHNAME system call 1-4, 2-3, 3-5, 8-20  
C\$GET\$OUTPUT\$CONNECTION system call 3-6, 4-1, 8-25  
C\$GET\$OUTPUT\$PATHNAME system call 2-3, 3-5, 8-31  
C\$GET\$PARAMATER system call 2-3, 3-10, 8-34  
changing the parsing buffer 3-15  
characters 8-11  
:CI: 8-45  
CLI 1-1, 2-2  
:CO: 8-45  
command connection 2-2, 5-1, 8-38  
    creating 8-4  
    deleting 8-8  
    sending commands 8-38  
command creation 7-1  
command line  
    interpreter 1-1, 2-2  
    parsing 3-1, 7-1  
    structure 3-1  
command name 3-1, 3-17, 8-13  
command processing system calls 5-1  
commands 1-1  
comment characters 3-3  
communicating with the terminal 4-3  
condition codes B-1  
configuration 9-1  
connections 4-1  
    input 8-15  
    output 8-25  
continuation characters 3-3, 8-38  
continuation lines 2-2  
Control-C handling 6-1  
CREATE\$COMMAND\$CONNECTION system call 2-3, 5-1, 8-4  
creating command connections 5-1, 8-4  
creating commands 7-1  
C\$SEND\$CO\$RESPONSE system call 2-2, 4-3, 8-45

## INDEX (continued)

C\$SEND\$COMMAND system call 2-2, 2-3, 3-3, 5-2, 8-38  
C\$SEND\$EO\$RESPONSE system call 4-3, 8-48  
C\$SET\$PARSE\$BUFFER system call 2-3, 3-16, 8-51  
customized initial program 2-3

DELETE\$COMMAND\$CONNECTION system call 2-3, 5-3, 8-8  
deleting command connections 5-3, 8-8  
dictionary of system calls 8-2  
displaying exception codes 4-4, 8-9  
dynamic memory size 7-4

errors B-1  
exception code formatting 4-4, 8-9  
exception codes B-1  
EXIT\$IO\$JOB system call 2-3, 7-2  
Extended I/O System 2-1  
extension objects 7-2

FORMAT\$EXCEPTION system call 4-4, 8-9

GET\$CHAR system call 3-15, 3-17, 8-11  
GET\$COMMAND\$NAME system call 3-17, 8-13  
GET\$EXCEPTION\$HANDLER system call 4-4  
GET\$INPUT\$CONNECTION system call 3-6, 4-1, 8-15  
GET\$INPUT\$PATHNAME system call 1-4, 2-3, 3-5, 8-20  
GET\$OUTPUT\$CONNECTION system call 3-6, 4-1, 8-25  
GET\$OUTPUT\$PATHNAME system call 2-3, 3-5, 8-31  
GET\$PARAMATER system call 2-3, 3-10, 8-34

I/O and message processing 4-1  
INCLUDE files 7-2  
initial program 1-1, 1-3, 2-2  
    customized 2-3  
    standard 2-2  
inpath-list 3-2  
input  
    connections 4-1, 8-15  
    pathnames 8-20  
INTEGER A-1  
interactive job 1-1

keyword 3-3, 3-11, 8-34

LINK86 command 7-3  
LOC86 7-4  
logon file 2-2

message processing system calls 4-1  
messages 8-9, 8-15, 8-26  
multi-access support 1-3, 2-1

nonresident configuration 9-2  
nonstandard command lines 3-13

INDEX (continued)

object code 7-3  
 outpath-list 3-2  
 output  
     connection 4-1, 8-25  
     pathnames 8-31  
 OVER preposition 3-2  
 overview 1-1  
  
 parameters 3-2, 8-34  
 parsing  
     buffer 3-1, 3-15, 8-51  
     commands 3-1, 7-1  
     input and output pathnames 3-5  
     nonstandard command lines 3-13  
     parameters 3-10  
 pathnames  
     input 8-20  
     output 8-31  
 POINTER A-1  
 preposition 3-2, 3-3, 3-11, 8-31, 8-35  
 :PROG: directory 2-2  
 program control 6-1  
  
 quoting characters (' or ") 3-4  
  
 R?LOGON file 2-2  
 ranges of exception codes B-2  
 regions 7-2  
 resident configuration 9-1  
 restricted system calls 7-2  
  
 S\$\$SPECIAL system call 6-2  
 SELECTOR A-1  
 semaphore 6-1  
 semicolon (;) 3-3  
 SEND\$CO\$RESPONSE system call 2-2, 4-3, 8-45  
 SEND\$COMMAND system call 2-2, 2-3, 3-3, 5-2, 8-38  
 SEND\$EO\$RESPONSE 4-3, 8-48  
 sending command lines to command connections 5-2  
 SET\$EXCEPTION\$HANDLER system call 4-4  
 SET\$PARSE\$BUFFER system call 2-3, 8-51  
 stack size 7-4  
 standard initial program 1-3, 2-2  
 STRING\$TABLE A-1, C-1  
 strings 3-6, A-1  
 structure of command lines 3-1  
 supplied commands 1-2  
 supporting multiple terminals 2-1  
 system call dictionary 8-2  
 system calls 1-2, 8-1  
     command-parsing 1-2, 3-1  
     command-processing 1-2, 5-1  
     I/O and message-processing 1-2, 4-1  
     program control 1-2, 6-1  
 system manager 1-3

INDEX (continued)

terminal

communications 4-3

messages 8-45, 8-48

terminating the command 7-2

TO preposition 3-2

TOKEN A-1

type definitions A-1

user IDs 1-3

value 3-3, 3-11, 8-34

wild-card characters 1-4, 3-8, 8-20

WORD A-1



## REQUEST FOR READER'S COMMENTS

Intel's Technical Publications Departments attempt to provide publications that meet the needs of all Intel product users. This form lets you participate directly in the publication process. Your comments will help us correct and improve our publications. Please take a few minutes to respond.

Please restrict your comments to the usability, accuracy, readability, organization, and completeness of this publication. If you have any comments on the product that this publication describes, please contact your Intel representative. If you wish to order publications, contact the Intel Literature Department (see page ii of this manual).

1. Please describe any errors you found in this publication (include page number).

---

---

---

---

---

---

---

2. Does the publication cover the information you expected or required? Please make suggestions for improvement.

---

---

---

---

---

---

---

3. Is this the right type of publication for your needs? Is it at the right level? What other types of publications are needed?

---

---

---

---

---

---

---

4. Did you have any difficulty understanding descriptions or wording? Where?

---

---

---

---

---

---

---

5. Please rate this publication on a scale of 1 to 5 (5 being the best rating). \_\_\_\_\_

NAME \_\_\_\_\_ DATE \_\_\_\_\_

TITLE \_\_\_\_\_

COMPANY NAME/DEPARTMENT \_\_\_\_\_

ADDRESS \_\_\_\_\_

CITY \_\_\_\_\_ STATE \_\_\_\_\_ ZIP CODE \_\_\_\_\_  
(COUNTRY)

Please check here if you require a written reply.

**WE'D LIKE YOUR COMMENTS . . .**

This document is one of a series describing Intel products. Your comments on the back of this form will help us produce better manuals. Each reply will be carefully reviewed by the responsible person. All comments and suggestions become the property of Intel Corporation.



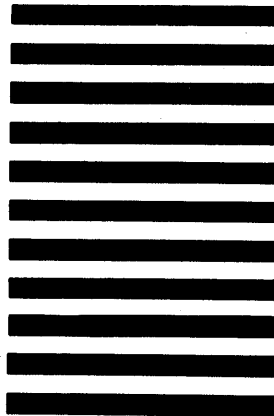
**NO POSTAGE  
NECESSARY  
IF MAILED  
IN THE  
UNITED STATES**

**BUSINESS REPLY MAIL**  
FIRST CLASS PERMIT NO. 79 BEAVERTON, OR

POSTAGE WILL BE PAID BY ADDRESSEE

**Intel Corporation**  
**5200 N.E. Elam Young Pkwy.**  
**Hillsboro, Oregon 97123**

**OMO Technical Publications**









**INTEL CORPORATION, 3065 Bowers Avenue, Santa Clara, California 95051 (408) 987-8080**

Printed in U.S.A.