iRMX™ 86 HUMAN INTERFACE REFERENCE MANUAL

CONTENTS

CONTENTS (continued)

TABLES

FIGURES

***

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) furnishes 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.

RESIDENT HUMAN INTERFACE COMMANDS

In addition to the code for the resident Human Interface, Intel has written a variety of commands which you can use with 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 isolate and identify the parameters in a command line. 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.

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 operator 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.

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 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. 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 CGETINPUT$PATHNAME and C$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 calling 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 the ampersand (&) mark in a command line and displays a different prompt (**) when a continuation line is required.
- Displays error messages in the event of certain operator errors.

This is the user 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.

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:

  CGETINPUT$PATHNAME
  CGETOUTPUT$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 CGETPARAMETER and CGETCHAR, it must first invoke the CSETPARSE$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 EXITIOJOB 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 EXITIOJOB 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:

- Defines the standard structure of command lines
- Describes the system calls used to parse commands having this structure
- Discusses how to switch from one parsing buffer to another parsing buffer
- Describes system calls 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 CGETINPUT$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 CGETOUTPUT$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 CGETOUTPUT$PATHNAME system call to process the outpath-list.

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 indicated 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 request.
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 CGETPARAMETER system call for more information).

You use the CGETPARAMETER system call to process the parameter.

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.

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, CGETCHAR, sees the parsing buffer as a 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, CGETCHAR 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 CGETINPUT$PATHNAME and C$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 CGETINPUT$PATHNAME reads the entire inpath-list (the list of pathnames separated by commas) into a buffer, moves the parsing pointer to the next parameter, and returns the first pathname to the command. Likewise, the first call to CGETOUTPUT$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 CGETINPUT$PATHNAME and C$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 CGETINPUT$PATHNAME obtains both input pathnames (A and B) and returns the first one (A) to the caller. The first call to CGETOUTPUT$PATHNAME obtains both output pathnames (C and D) and returns the first one (C) to the caller. CGETOUTPUT$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 CGETINPUT$PATHNAME and C$GET$OUTPUT$PATHNAME in a particular order. To use these system calls effectively, obey the following rules:

1. Always call CGETINPUT$PATHNAME to obtain the input pathname before calling CGETOUTPUT$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, CGETOUTPUT$PATHNAME cannot determine the output pathname until CGETINPUT$PATHNAME determines the corresponding input pathname.

2. Always alternate your calls to CGETINPUT$PATHNAME and C$GET$OUTPUT$PATHNAME. This is necessary to handle wild-card characters and lists of pathnames. If you invoke two calls to CGETINPUT$PATHNAME without an intermediate call to CGETOUTPUT$PATHNAME, you will not be able to obtain the first output pathname. Similarly, if you invoke two calls to CGETOUTPUT$PATHNAME without an intermediate call to CGETINPUT$PATHNAME, the second call returns invalid information.

CGETINPUT$PATHNAME and C$GET$OUTPUT$PATHNAME return the pathnames in the form of iRMX 86 strings. Each string is a group of bytes in which the first byte contains the number of ASCII bytes that follow. For these system calls, the remaining bytes in the string contain the pathname. If CGETINPUT$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 CGETINPUT$PATHNAME and C$GET$OUTPUT$PATHNAME to obtain the input file and corresponding output file, you can use the system calls CGETINPUT$CONNECTION and C$GET$OUTPUT$CONNECTION to obtain connections to those files. Chapter 4 contains more information about CGETINPUT$CONNECTION and C$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 CGETINPUT$PATHNAME and C$GET$OUTPUT$PATHNAME in its command-line parsing (it also uses CGETINPUT$CONNECTION and C$GET$OUTPUT$CONNECTION to obtain connections to the files. This command is a partial example of a COPY command that you could implement.

```
/*****************************************************************************
*   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. CGETINPUT$PATHNAME And C$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. CGETINPUT$PATHNAME And C$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
```

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

CGETINPUT$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 of the next file that matches. For example, if an input pathname is:

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

CGETINPUT$PATHNAME searchs the :PROG:PLM directory and returns the pathname of the next file that begins with the letter "A."

CGETOUTPUT$PATHNAME generates wild cards. Each time you call it, it compares the wild-carded output pathname with the wild-carded input pathname and with the most recent pathname returned by CGETINPUT$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
A11       B11
ADAM      C11
```

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 CGETINPUT$PATHNAME and C$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, CGETINPUT$PATHNAME and C$GET$OUTPUT$PATHNAME return pathnames as follows:

Pathname list returned by CGETINPUT$PATHNAME	Corresponding pathname list returned by CGETOUTPUT$PATHNAME
ALPHA	CLPHA
A11	C11
ADAM	CDAM
BETA	DETA
B11	D11

PARSING OTHER PARAMETERS

The CGETPARAMETER 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 CGETPARAMETER 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 CGETINPUT$PATHNAME and C$GET$OUTPUT$PATHNAME which handle wild-card characters automatically. For example, suppose a command line contains the pathname:

```
FILE*
```

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

However, if you use CGETPARAMETER 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, CGETPARAMETER 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:

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. CGETPARAMETER 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, CGETPARAMETER treats the values as a single value, returning them in single string. For example, in the following value-list:

```
A,(B,C,D),E
```

CGETPARAMETER 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 CGETPARAMETER 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.                                                        *
*****************************************************************************/

prog1: DO;

$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$buf),
        excep WORD,
        (size$val, i) WORD;
```

Figure 3-2. CGETPARAMETER Example

```
/* 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. CGETPARAMETER Example (continued)

PARSING NONSTANDARD COMMAND LINES

If the command line you write follows the recommended structure described earlier in this chapter, you can use CGETINPUT$PATHNAME, C$GET$OUTPUT$PATHNAME, and CGETPARAMETER 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. With this convention, commands always call CGETINPUT$PATHNAME and C$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 CGETINPUT$PATHNAME and C$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 CGETPARAMETER to parse the first and second parameters. Then it could use CGETINPUT$PATHNAME and C$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 CGETPARAMETER to parse parameters. Then, whenever it received a parameter whose value was "FROM", it could call another portion of code that used CGETINPUT$PATHNAME and C$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.

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 CGETINPUT$PATHNAME, C$GET$OUTPUT$PATHNAME, and CGETPARAMETER 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 CGETCHAR 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.

CGETCHAR 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 CGETCHAR you can enforce any command syntax you choose.

Because CGETCHAR moves the pointer character by character, not parameter by parameter, you should take care when using CGETCHAR in the same program with CGETINPUT$PATHNAME, C$GET$OUTPUT$PATHNAME, and CGETPARAMETER. You must ensure that CGETCHAR 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.

SWITCHING TO ANOTHER 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.

Some commands might require the ability to parse additional lines of text (for example, an editor needs to parse individual editor commands) after the original command invocation. 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, CSETPARSE$BUFFER, switches the parsing buffer and sets the parsing pointer to the beginning of the buffer.

One of the parameters of the CSETPARSE$BUFFER system call (buff$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 CSETPARSE$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 (CGETINPUT$PATHNAME and C$GET$OUTPUT$PATHNAME) are not affected by calls to CSETPARSE$BUFFER. These calls always obtain pathnames from the original parsing buffer (the command line).

When you establish a new parsing buffer, CSETPARSE$BUFFER sets the parsing pointer to the beginning of the buffer. This allows you to use one buffer for parsing many lines of text. For example, suppose your command has several sub-commands. Each time the operator enters a sub-command, your command reads the sub-command into a buffer, calls CSETPARSE$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$RESPONSE, C$SENDEORESPONSE, or an Extended I/O System call).

2. Call CSETPARSE$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 CGETPARAMETER or CGETCHAR 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.

If you specify a zero value for the buff$p parameter of C$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 CSETPARSE$BUFFER, the parsing pointer moves to the start of the buffer. Therefore, you lose your place in the buffer. However, CSETPARSE$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 CSETPARSE$BUFFER), you can use this value to move the pointer to its previous position.

One way to do this is to use the CGETCHAR system call to move the parsing pointer back to its previous position. After switching back to the original buffer, call CGETCHAR the number of times specified in the offset parameter of the first CSETPARSE$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 CSETPARSE$BUFFER the first time, you can specify the buff$p parameter to point to the first element of the array (CHAR(0), for example). Then, when you switch parsing buffers, CSETPARSE$BUFFER returns, in the offset parameter, the number of bytes already parsed. When you switch back to the first parsing buffer, you can use this offset value as an index into the array; that is, have the buff$p 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 CGETCOMMAND$NAME.

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

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 CGETCOMMAND$NAME and the ampersand pathname separator (&) to access the directory in which it resides. For example, if "command-name" is the name received from CGETCOMMAND$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: CGETINPUT$CONNECTION and C$GET$OUTPUT$CONNECTION. These system calls are structured so that you can use the output from CGETINPUT$PATHNAME and C$GET$OUTPUT$PATHNAME as input to these system calls.

USING CGETINPUT$CONNECTION

CGETINPUT$CONNECTION obtains a connection to a file and opens that connection for reading. One of the parameters of CGETINPUT$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 CGETINPUT$PATHNAME or it can be the pathname of any other file to which you want a connection. If CGETINPUT$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, CGETINPUT$CONNECTION displays the following message:

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

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

Because CGETINPUT$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 CGETOUTPUT$CONNECTION

CGETOUTPUT$CONNECTION obtains a connection to a file and opens that connection for writing. As in the case of CGETINPUT$CONNECTION, one of the parameters of CGETOUTPUT$CONNECTION is a pointer to a string containing the pathname of the file for which a connection is sought.

This pathname can be the pathname returned by CGETOUTPUT$PATHNAME or it can be the pathname of any other file to which you want a connection. There is another parameter in CGETOUTPUT$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, CGETOUTPUT$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, CGETOUTPUT$CONNECTION obtains the connection without prompting the operator for permission.

If you specify the AFTER preposition, CGETOUTPUT$CONNECTION obtains the connection without prompting the operator for permission. It also seeks to the end of file before returning control. Thus any information you write to the file will not overwrite the existing information. This is unlike TO and OVER which cause CGETOUTPUT$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 CGETOUTPUT$CONNECTION cannot obtain a connection to the file, CGETOUTPUT$CONNECTION returns an exception code and displays a message at the operator's terminal. Refer to the description of CGETOUTPUT$CONNECTION in Chapter 7 for more information.

EXAMPLE PROGRAM SCENARIO

A normal scenario for using CGETINPUT$CONNECTION and C$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$RESPONSE and C$SENDEORESPONSE. 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:

- Send a message and receive a message
- Send a message without waiting to receive a message
- 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 a difference between C$SEND$CO$RESPONSE and C$SENDEORESPONSE. 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$RESPONSE, C$SENDEORESPONSE always reads information from and writes information to the operator's terminal. Input and output cannot be redirected with C$SEND$EO$RESPONSE.

C$SEND$EO$RESPONSE is 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, when used by all the tasks which communicate with the terminal, prevents the operator from receiving several requests for information before being able to respond to the first one.

FORMATTING MESSAGES BASED ON EXCEPTION CODES

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 effect of making these system calls is the same as that produced by an operator entering a command line at the terminal. 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.

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. 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.

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 command    *
*   COPY :F1:OLD to :F1:NEW 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 (hgtocn.ext)

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

Figure 5-1. Command Connection Example

```
                    .
                    .
                    .

/* 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. 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.

You can use the S$SPECIAL system call of the Extended I/O System to replace 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 task 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. Once the job for the initial command is deleted by the Human Interface, then Control-C once again becomes the default method for program control. The Human Interface reactivates Control-C by resetting a semaphore when the original command finishes. For example, once the text editor we used as an example terminates, then the Human Interface resets the semaphore so that Control-C becomes active.

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 S$ATTACH$FILE to obtain a connection to the terminal. Use logical name :CI: as the pathname parameter.

4. Call S$OPEN to open the connection to the terminal for reading only (mode 1).

5. If you plan to use a Control-C task, have the program call CREATE$TASK to start the Control-C task.

6. Call S$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.

To 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 your code services 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.

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 CGETINPUT$PATHNAME to obtain the first input pathname.
2. Call CGETOUTPUT$PATHNAME to obtain the preposition and first output pathname.
3. Call CGETPARAMETER as many times as necessary to get all the parameters.
4. Do until no more input pathnames remain:
    a. Call CGETINPUT$CONNECTION to obtain a connection to the input file.
    b. Call CGETOUTPUT$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.

d. Call S$DELETE$CONNECTION (Extended I/O System call) to delete the connections to the input and output files.

e. Call CGETINPUT$PATHNAME and C$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 lock up when the command finishes 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 EXITIOJOB (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 EXITIOJOB, 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.

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)
```

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).

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 monitor 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 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 several 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."

Table 8-1. System Call Dictionary

System Call	Synopsis	Page
	I/O Processing Calls	
CGETINPUT$CONNECTION	Return an EIOS connection for the specified input file.	8-15
CGETOUTPUT$CONNECTION	Return an EIOS connection for the specified output file.	8-25
	Command Parsing Calls	
CGETCHAR	Get a character from the command line	8-11
CGETINPUT$PATHNAME	Parse the command line and return an input pathname.	8-20
CGETPARAMETER	Parse the command line for the next parameter and return it as a keyword name and a value.	8-34
CGETOUTPUT$PATHNAME	Parse the command line and return an output pathname.	8-31
CSETPARSE$BUFFER	Parse a buffer other than the current command line.	8-51
CGETCOMMAND$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

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$co,
                                              flags, except$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.
flags	A WORD used to indicate that the Human Interface should return an E$ERROROUTPUT exception code if the system call C$SEND$EO$RESPONSE is used by any task. If the user wants the exception code, then the parameter is set to one (1); otherwise, the parameter must equal 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.

A possible application where the parameter "flags" might be set to one is when you want to write a custom CLI to perform batch jobs in the background. When any of the background batch jobs attempt to communicate with the terminal through C$SEND$EO$RESPONSE, the Human Interface issues an exception code. In this way, the Human Interface keeps all the jobs in the background. Note--the Human Interface CLI does not provide resident background or batch processing capability.

EXCEPTION CODES

E$OK	No exceptional conditions were encountered.
E$ALREADY$ATTACHED	While creating a STREAM file, the Extended I/O System was unable to attach the :STREAM: device because another task had already invoked a Basic I/O system call to attach the :STREAM: device.
E$CONTEXT	At least one of the following is true: • Two command connections were being created simultaneously by two tasks in the same job. • 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.)
EDEVDETACHING	The :STREAM: device, the default$ci device, or the default$co device was in the process of being detached.
E$DEVFD	The Extended I/O System attempted the physical attachment of the :STREAM: device. This device had formerly been only logically attached. In the process, the Extended I/O System found that the device and the device driver specified in the logical attachment were incompatible. The Operating System would not have returned this exception code if the :STREAM: device had been properly configured.
E$EXIST	The default$ci or default$co parameter is not a token for an existing job.
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$co connection. However, the request for information resulted in an invalid file driver request.
E$INVALID$FNODE	The fnode associated with the specified file is invalid. Delete the file.
E$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$LIMIT	At least one of the following is true: • The object directory of the calling task's job has already reached the maximum object directory size. • The calling task's job has exceeded its object limit. • The calling task's job (or that job's default user object) is already involved in 255 (decimal) I/O operations. • The calling task's job is not I/O job. (Refer to the iRMX 86 EXTENDED I/O SYSTEM REFERENCE MANUAL for information about I/O jobs.)
ELOGNAME$NEXIST	The call was unable to find the logical name :STREAM: in the object directories of the local job, the global job, or the root job.
E$MEM	The memory available to the calling task's job is not sufficient to complete the call.
ENOPREFIX	The calling task's job does not have a valid default prefix.
ENOTCONNECTION	The default$ci or default$co parameter is a token for an object, not a connection to a file.
ENOTLOG$NAME	The logical name :STREAM: refers to an object that is not a file or device connection.
ENOUSER	The calling task's job does not have a valid default user object.

E$PARAM	The system call forced the Extended I/O System to attempt the physical attachment of the :STREAM: device, which had formerly been only logically attached. In the process, the Extended I/O System found that the stream file driver is not properly configured into your system, so the physical attachment is not possible.
E$SUPPORT	The default$ci or default$co connection was not created by this job.

C$DELETE$COMMAND$CONNECTION

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

```
CALL RQ$C$DELETE$COMMAND$CONNECTION(command$conn, except$ptr);
```

INPUT PARAMETER

command$conn	A TOKEN for a valid command connection.

OUTPUT PARAMETER

except$ptr	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

E$OK	No exceptional conditions were encountered.
E$EXIST	The command$conn parameter is not a token for an existing object.
E$TYPE	The command$conn parameter is a token for 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 undefined exception code value was specified.
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.

CGETCHAR

CGETCHAR, 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 CGETCHAR system call gets a single character from that buffer and moves the parsing pointer to the next character. Consecutive calls to CGETCHAR return consecutive characters from the parsing buffer.

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. • The object directory of the calling task's job has already reached the maximum object directory size. • The calling task's job has exceeded its object limit.

- 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 available to the calling task's job is not sufficient to complete the call.

CGETCOMMAND$NAME

CGETCOMMAND$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 CGETCOMMAND$NAME returns this information. This information is available to each command and is stored in a buffer that is separate from the parsing buffer. Therefore, calling CGETCOMMAND$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 CGETCOMMAND$NAME also includes the directory name (such as :SYSTEM:, :PROG:, or :$:) 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.

E$PATHNAME$SYNTAX	The specified pathname contains invalid characters.
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.

CGETINPUT$CONNECTION

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

```
connection = RQ$C$GET$INPUT$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

CGETINPUT$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 (The buffer size may be different than the default value of 1024 bytes.)

CGETINPUT$CONNECTION causes an error message to be displayed at the operator's terminal (:CO:) whenever the Operating System encounters an exceptional condition. The exceptional condition that triggers the error message can either be one of those listed for CGETINPUT$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.

- <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 CGETINPUT$CONNECTION attempted to obtain the input connection. The <exception value> and <exception mnemonic> portions of the message indicate the exception code encountered. Refer to "Exception Codes" in this call description and to the descriptions of S$ATTACH$FILE and S$OPEN in the iRMX 86 EXTENDED I/O SYSTEM REFERENCE MANUAL.

EXCEPTION CODES

E$OK	No exceptional conditions were encountered.
E$ALREADY$-ATTACHED	The device containing the file specified in the path$name$p parameter is already attached.
E$CONTEXT	At least one of the following is true: • 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.) • The calling task's job was not created by the Human Interface.
EDEVDETACHING	The device specified in the path$name$p parameter is in the process of being detached.

E$DEVFD	The call attempted the physical attachment of a device that had formerly been only logically attached. In the process, the call found that the device and the device driver specified in the logical attachment were incompatible.
E$EXIST	The specified device does not exist.
E$FACCESS	The specified connection does not have read access to the file.
E$FNEXIST	At least one of the following is true: • The target file does not exist or is marked for deletion. • While attaching the file pointed to by the path$name$p parameter, the call attempted the physical attachment of the device as a named device. It could not complete this process because the device specified when the logical attachment was made was not defined during configuration.
E$FTYPE	The path pointed to by the path$name$p parameter contained a file name that should have been the name of a directory, but is not. (Except for the last file, each file in a pathname must be a named directory.
E$ILLVOL	The call attempted the physical attachment of the specified device as a named device. This device had formerly been only logically attached. The call found that the volume did not contain named files. This prevented the call from completing physical attachment because the named file driver was requested during logical attachment.
E$INVALID$-FNODE	The fnode for the specified file is invalid, so the file must be deleted.
EIOHARD	While attempting to access the file specified in the path$name$p parameter, the call detected a hard I/O error. This means that another call is probably useless.
E$IOMEM	While attempting to create a connection, the call needed memory from the Basic I/O subsystem's memory pool. However, the Basic I/O System job does not currently have a block of memory large enough to allow this call to run to completion.
EIOOPRINT	While attempting to access the file specified in the path$name$p parameter, the call found that the device was off-line. Operator intervention is required when given this code.

EIOSOFT	While attempting to access the file specified in the path$name$p parameter, the call detected a soft I/O error. It tried the operation again but was unsuccessful. Another try might be successful.
EIOUNCLASS	An unknown type of I/O error occurred while this call tried to access the file given in the path$name$p parameter.
E$LIMIT	At least one of the following is true: • The calling task's job or the job's default user object is already involved in 255 (decimal) I/O operations. • The calling task's job was not created by the Human Interface.
ELOGNAME$-NEXIST	The pathname for the specified device contains an explicit logical name. The call was unable to find this name in the object directories of the local job, the global job, or the root job.
ELOGNAME$-SYNTAX	The pathname pointed to by the path$name$p parameter contains a logical name. However, the logical name contains an unmatched colon, is longer than 12 characters, has zero (0) characters, or contains invalid characters.
E$MEDIA	The specified device was off-line.
E$MEM	The memory available to the calling task's job is not sufficent to complete the call.
ENOPREFIX	The calling task's job does not have a valid default prefix.
ENOTLOG$NAME	The logical name specified by the path$name$p parameter does not refer to a file or device connection.
ENOUSER	The calling task's job does not have a valid default user.
E$PARAM	At least one of the following is true: • The system call forced the Extended I/O System to attempt the physical attachment of the device referenced by the path$name$p parameter. This device had formerly been only logically attached. In the process, the Extended I/O System found that the logical attachment referred to a file driver (named, physical, or stream) that is not configured into your system, so the physical attachment is not possible.

	• The connection to the specified file cannot be opened for reading.
E$PATHNAME$-SYNTAX	The specified pathname contains invalid characters.
E$SHARE	The files sharing attribute currently does not allow new connections to the file to be opened for reading.
E$STREAM$SPECIAL	The call attempted to attach a stream file and in so doing issued an invalid stream file request.

CGETINPUT$PATHNAME

CGETINPUT$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 CGETINPUT$PATHNAME retrieves the entire input pathname list and moves the parsing pointer to the next parameter. CGETINPUT$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 CGETINPUT$PATHNAME return additional pathnames from the input pathname list but do not move the parsing pointer. CGETINPUT$PATHNAME denotes the end of the pathname list by returning a zero-length string.

CGETINPUT$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 CGETINPUT$PATHNAME can be used for any purpose. However, it is most often used in a call to CGETINPUT$CONNECTION, to obtain a connection.

EXCEPTION CODES

E$OK	No exceptional conditions were encountered.
E$ALREADY$-ATTACHED	The device containing the file pointed to by the path$name$p parameter is already attached.
E$CONTEXT	At least one of the following is true: • 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 task called CGETOUTPUT$PATHNAME before calling CGETINPUT$PATHNAME.
EDEVDETACHING	The device pointed to by the path$name$p parameter is in the process of being detached.
E$DEVFD	The Extended I/O System attempted the physical attachment of a device that had formerly been only logically attached. In the process, the Extended I/O System found that the device and the device driver specified in the logical attachment were incompatible.
E$EXIST	At least one of the following is true: • The connection to the parent directory of the file pointed to by the path$name$p parameter, is not a token for the existing job. • The calling task's job was not created by the Human Interface.
E$FACCESS	The connection used to open the directory does not have read access to the directory.
E$FLUSHING	The device containing the directory was in the process of being detached.

E$FNEXIST	At least one of the following is true: • The target file does not exist or is marked for deletion. • While attaching the parent directory of the file pointed to by the path$name$p parameter, the I/O System attempted the physical attachment of the device as a named device. It could not complete this process because the device specified when the logical attachment was made was not defined during configuration.
E$FTYPE	The path pointed to by the path$name$p parameter contained a file name that should have been the name of a directory, but is not. (Except for the last file, each file in a pathname must be a named directory.
E$IFDR	The specified file is a stream or physical file.
E$ILLVOL	The call attempted the physical attachment of the specified device as a named device. This device had formerly been only logically attached. The call found that the volume did not contain named files. This prevented the call from completing physical attachment because the named file driver was requested during logical attachment.
E$INVALID$-FNODE	The fnode for the specified file is invalid, so the file must be deleted.
EIOHARD	While attempting to access the parent directory of the file pointed to by the path$name$p parameter, the call detected a hard I/O error. This means that another call is probably useless.
E$IOMEM	While attempting to create a connection, this call needed memory from the Basic I/O subsystem's memory pool. However, the Basic I/O System job does not currently have a block of memory large enough to allow this call to run to completion.
EIOOPRINT	While attempting to access the parent directory of the file pointed to by the path$name$p parameter, this call detected that the device was off-line. Operator intervention is required. C$FORMAT$EXCEPTION returns the value EIONOT$READY for this code.

EIOSOFT	While attempting to access the parent directory of the file pointed to by the path$name$p parameter, this call detected a soft I/O error. It tried the operation again, but was unsuccessful. Another try might be successful.
EIOUNCLASS	An unknown type of I/O error occurred while this call tried to access the parent directory of the file pointed to by the path$name$p parameter.
E$LIMIT	At least one of the following is true: • The calling task's job has already reached its object limit. • The calling task's job or the job's default user object is already involved in 255 (decimal) I/O operations. • The calling task's job was not created by the Human Interface.
E$LIST	The last value of the input pathname list is missing. For example, "ABLE,BAKER," has no value following the second comma.
ELOGNAME$-NEXIST	The pathname for the specified device contains an explicit logical name. The call was unable to find this name in the object directory of the local job, the global job, or the root job.
ELOGNAME$-SYNTAX	The pathname pointed to by the path$name$p parameter contains a logical name. However, the logical name contains an unmatched colon, is longer than 12 characters, has zero (0) characters, or contains invalid characters.
E$MEDIA	The specified device was off-line.
E$MEM	The memory available to the calling task's job is not sufficient to complete the call.
ENOPREFIX	The calling task's job does not have a valid default prefix.
ENOTLOG$NAME	The logical name specified by the path$name$p parameter does not refer to a file or device connection.
ENOUSER	The calling task's job does not have a valid default user object.

E$PARAM	At least one of the following is true: • The Extended I/O System attempted the physical attachment of the device pointed to by the path$name$p parameter. This device had formerly been only logically attached. In the process, the Extended I/O System found that the logical attachment referred to a file driver (named, physical, or stream) that is not configured into your system, so the physical attachment is not possible. • The connection to the parent directory cannot be opened for reading.
E$PARSE$TABLES	The call detected an error in an internal table used by the Human Interface.
E$PATHNAME$-SYNTAX	The specified pathname contains invalid characters.
E$SHARE	The connection to the parent directory cannot be opened for reading.
E$STREAM$-SPECIAL	The Extended I/O System attempted to attach a stream file and in so doing issued an invalid stream file request.
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	This call 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.

CGETOUTPUT$CONNECTION

CGETOUTPUT$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:

Value	Meaning
0	Use same preposition as was returned by the last CGETOUTPUT$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

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

- Write only
- Accessible to all

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

<pathname>, already exists, OVERWRITE?

If the operator enters Y, y, R, or r, CGETOUTPUT$CONNECTION returns a connection to the existing file, allowing the command to write over the file. Any other response causes CGETOUTPUT$CONNECTION to generate an E$FILEACCESS exception code.

CGETOUTPUT$CONNECTION causes an error message to be displayed at the operator's terminal (:CO:) whenever an exceptional condition occurs. The exceptional condition that triggers the error message can be either one of those listed for CGETOUTPUT$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 an existing file.

- <pathname>, directory ADD entry access required

  The user does not have add entry access to the parent 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 CGETOUTPUT$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$OK	No exceptional conditions were encountered.
E$ALREADY$-ATTACHED	The Extended I/O System was unable to attach the device containing the file because the Basic I/O System has already attached the device.
E$CONTEXT	The calling task's job was not created by the Human Interface.
EDEVDETACHING	The device referred to by the path$name$p parameter was in the process of being detached.
E$DEVFD	The call attempted the physical attachment of a device that had formerly been only logically attached. In the process, the call found that the device and the device driver specified in the logical attachment were incompatible.
E$EXIST	The connection parameter for the device containing that file is not a token for an existing object.
E$FACCESS	At least one of the following is true: • The default user for the calling task's job did not have update access to an existing file and/or add-entry access to the parent directory. • The TO or OVER preposition was specified and the default user for the calling task's job did not have the ability to truncate the file.

E$FNEXIST	At least one of the following is true: • The target file does not exist or is marked for deletion. • While attaching the file pointed to by the path$name$p parameter, the Extended I/O System attempted the physical attachment of the device as a named device. It could not complete this process because the device specified when the logical attachment was made was not defined during configuration.
E$FTYPE	The path pointed to by the path$name$p parameter contained a file name that should have been the name of a directory, but is not. (Except for the last file, each file in a pathname must be a named directory.
E$IFDR	The call requested information about the specified file, but the request was an invalid file driver request.
E$ILLVOL	The call attempted the physical attachment of the specified device as a named device. This device had formerly been only logically attached. The call found that the volume did not contain named files. This prevented the call from completing physical attachment because the named file driver was requested during logical attachment.
E$INVALID$-FNODE	The fnode for the specified file is invalid, so the file must be deleted.
EIOHARD	While attempting to access the file specified in the path$name$p parameter, the call detected a hard I/O error. This means that another try is probably useless.
E$IOMEM	While attempting to create a connection, this call needed memory from the Basic I/O subsystem's memory pool. However, the Basic I/O System job does not currently have a block of memory large enough to allow this call to run to completion.
EIOOPRINT	While attempting to access the file specified in the path$name$p parameter, the call detected that the device was off-line. Operator intervention is required.

EIOSOFT	While attempting to access the file specified in the path$name$p parameter, the call detected a soft I/O error. It tried the operation again but was unsuccessful. Another try might be successful.
EIOUNCLASS	An unknown type of I/O error occurred while this call tried to access the file given in the path$name$p parameter.
EIOWRPROT	While attempting to obtain an input connection to the file specified in the path$name$p parameter, this call found that the volume containing the file is write-protected.
E$LIMIT	At least one of the following is true: • The calling task's job or the job's default user object is already involved in 255 (decimal) I/O operations. • The calling task's job is not an I/O job. (Refer to the iRMX 86 EXTENDED I/O SYSTEM REFERENCE MANUAL for more information about I/O jobs.)
ELOGNAME$-NEXIST	The specified pathname contains an explicit logical name. The call was unable to find this name in the object directory of the local job, the global job, or the root job.
ELOGNAME$-SYNTAX	The pathname pointed to by the path$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 specified device was off-line.
E$MEM	The memory available to the calling task's job is not sufficient to complete the call.
ENOPREFIX	The calling task's job does not have a valid default prefix.
ENOTLOG$NAME	The logical name specified by the path$name$p parameter does not refer to a file or device connection.
ENOUSER	The calling task's job does not have a valid default 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$p parameter. The device had formerly been only logically attached. In the process, the Extended I/O System found that the logical attachment referred to a file driver (named, physical, or stream) that is not configured into your system, so the physical attachment is not possible.
E$PATHNAME$-SYNTAX	The specified pathname contains invalid characters.
E$PREPOSITION	One of the following is true: • 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 CGETOUTPUT$PATHNAME call. However, this is the first call to CGETOUTPUT$PATHNAME.
E$SHARE	The new connection cannot be opened for writing.
E$SPACE	One of the following is true: • The volume is full. • The volume already contains the maximum number of files.
E$STREAM$SPECIAL	The Extended I/O System attempted to attach a stream file and in so doing issued an invalid stream file request.

CGETOUTPUT$PATHNAME

CGETOUTPUT$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 CGETOUTPUT$PATHNAME encountered. You can pass this value to CGETOUTPUT$CONNECTION when obtaining an output connection to the file. The value will be one of the following:

Value	Meaning
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.

DESCRIPTION

You should not call CGETOUTPUT$PATHNAME before first calling CGETINPUT$PATHNAME.

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

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

The pathname returned by CGETOUTPUT$PATHNAME can be used for any purpose. However, it is most often used in a call to CGETOUTPUT$CONNECTION to obtain a connection to the file. In such a case, CGETOUTPUT$CONNECTION processes the TO/OVER/AFTER preposition. If the pathname is used as input to a system call other than CGETOUTPUT$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 default output string pointed to by default$output$p contained an invalid preposition or pathname.

E$LIMIT	At least one of the following is true: • The calling task's job has already reached its limit. • The calling task's job was not created by the Human Interface.
E$MEM	The memory available to the calling task's job is not sufficient to complete the call.
E$PATHNAME$-SYNTAX	The specified pathname contains invalid characters.
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$-LISTS	The numbers of files in the input and output lists are not same.
E$WILDCARD	The output pathname contains an invalid wild-card specification.

CGETPARAMETER

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 CGETPARAMETER 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

CGETPARAMETER 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 3.

CGETPARAMETER 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, CGETPARAMETER cannot determine whether groups of characters separated by spaces are separate parameters or a single parameter that uses a preposition. CGETPARAMETER 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 CGETPARAMETER 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, CGETPARAMETER 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 call found a continuation character in the parse buffer. Command lines should not contain continuation characters.

E$LIMIT — At least one of the following is true:

- The calling task's job has already reached its object limit.
- 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 is true:

- 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:

Value	Comments
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 or set off by commas. A,B,(C,E),F is valid.

E$LITERAL — The call found a literal (quoted string) in the parsing buffer with no closing quote. This condition should not occur in the command line buffer.

E$MEM — The memory available to the calling task's job is not sufficient to complete the call.

E$PARAM — The predict$list$p parameter pointed to a string table, but the index$p parameter was set to zero (0).

E$PARSE$TABLES	The call found an error in an internal table used by the Human Interface.
E$SEPARATOR	The call found an invalid command separator in the parsing buffer. This condition should not occur in the command line buffer. The following is a list of invalid command separators: ><, <>, ||, |, [, and].
E$STRING	The string to be returned as the parameter name or one of the parameter values exceeds the length limit of 255 characters.
E$STRING$BUFFER	The string to be returned as the parameter name or one of the parameter values exceeds the buffer size provided 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

command$conn	A TOKEN for the command connection that receives the command line.
line$p	A POINTER to a STRING containing a command line to execute.

OUTPUT PARAMETERS

command$except$ptr	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 except$ptr.
except$ptr	A POINTER to a WORD in which the Human Interface returns a condition code indicating the status of the C$SEND$COMMAND 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$OK	No exceptional conditions were encountered.
E$ALREADY$-ATTACHED	The Extended I/O System was unable to attach the device containing the object file because the Basic I/O System has already attached the device.
EBADGROUP	The object file represented by the command's pathname contained an invalid group definition record.
EBADHEADER	The object file represented by the command's pathname does not begin with a header record for a loadable object module.
EBADSEGMENT	The object file represented by the command's pathname contains an invalid segment definition record.
E$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	The calling task's job was not created by the Human Interface.

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.
EDEVDETACHING	The device containing the object file was in the process of being detached.
E$DEVFD	The Extended I/O System attempted the physical attachment of a device that had formerly been only logically attached. In the process, the Extended I/O System found that the device and the device driver specified in the logical attachment were incompatible.
E$EOF	The Application Loader encountered an unexpected end of file on the object file represented by the command's pathname.
E$EXIST	At least one of the following is true: • The call detached the device containing the object file before completing the loading operation. • The command$conn parameter is not the token for a command connection.
E$FACCESS	The default user for the calling task's job does not have read access to the object file.
E$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 device containing the object file was being detached.

E$FNEXIST	At least one of the following is true: • The file in the command's pathname is either marked for deletion or does not exist. • While attaching the file specified in the line$p parameter, the Extended I/O System attempted the physical attachment of the device as a named device. It could not complete this process because the device specified when the logical attachment was made was not defined during configuration.
E$FTYPE	The path pointed to by the path$name$p parameter contained a file name that should have been the name of a directory, but is not. (Except for the last file, each file in a pathname must be a named directory.
E$ILLVOL	The call attempted the physical attachment of the specified device as a named device. This device had formerly been only logically attached. The call found that the volume did not contain named files. This prevented the call from completing physical attachment because the named file driver was requested during logical attachment.
E$INVALID$-FNODE	The fnode for the specified file is invalid, so the file must be deleted.
EIOHARD	While attempting to access the object file, this call detected a hard I/O error. This means that another try is probably useless.
E$IOMEM	The Basic I/O System does not currently have a block of memory large enough to allow the Human Interface to create the connection necessary to allow this call to run to completion.
EIOOPRINT	While attempting to access the object file, this call found that the device was off-line. Operator intervention is required. C$FORMAT$EXCEPTION returns the value EIONOT$READY when given this code.
EIOSOFT	While attempting to access the object file, this call detected a soft I/O error. It tried again, but was not successful. Another try might be successful.
EIOUNCLASS	An unknown type of I/O error occurred while this call tried to access the object file.

EIOWRPROT	While attempting to obtain an input connection to the object file, the call found that the volume containing the file is write-protected.
E$LIMIT	At least one of the following is true: • The calling task's job has already reached its object limit. • The calling task's job , or the job's default user object, is already involved in 255 (decimal) I/O operations. • The new I/O job, or its default user, is already involved in 255 (decimal) I/O operations. • The calling task's job is not an I/O job. Refer to the iRMX 86 EXTENDED I/O SYSTEM REFERENCE MANUAL for information about I/O jobs.
E$LITERAL	The call found a literal (quoted string) with no closing quote while scanning the contents of the command line pointed to by the line$p parameter.
ELOGNAME$-NEXIST	The command's pathname contains an explicit logical name but the call was unable to find this name in the object directory of the local job, the global job, or the root job.
ELOGNAME$-SYNTAX	The pathname pointed to by the path$name$p parameter contains a logical name. However, the logical name contains an unmatched colon, is longer than 12 characters, has zero (0) characters, or contains invalid characters.
E$MEDIA	The device containing the object file was off-line.
E$MEM	The memory available to the calling task's job, the new I/O job, or the Basic I/O System job is not sufficient to complete the call.
ENOLOADER$MEM	At least one of the following is true: • 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.

ENOMEM	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.
ENOPREFIX	The calling task's job does not have a valid default prefix.
ENOSTART	The object file represented by the command-pathname does not specify the entry point for the program being loaded.
ENOTCONNECTION	The default$ci or default$co parameter is a token for an object that is not a file connection.
ENOTLOG$NAME	The command pathname contains a logical name. The logical name of an object that is neither a device connection nor a file connection.
ENOUSER	The calling task's job does not have a valid default user.
E$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 call found an error in an internal table.
E$PATHNAME$-SYNTAX	The command's pathname contains invalid characters.
ERECFORMAT	At least one record in the object file contains a record format error.
ERECLENGTH	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.
ERECTYPE	At least one of the following is true: • At least one record in the file being loaded is of a type that the Application Loader cannot process. • The Application Loader has encountered records in a sequence that it cannot process.

ESEGBOUNDS	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$SEPARATOR	The call found an invalid separator while scanning the command line. The following is a list of the invalid command separators: ><, <>, ||, |, [, and].
E$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$TIME	The calling task's job was not created by the Human Interface.
E$TYPE	The command$conn parameter is 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:

message$p	response$max	Action
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:, output from other tasks can be displayed at :CO: while the system waits for a response from :CI:.

The main distinction between C$SEND$CO$RESPONSE and C$SENDEORESPONSE 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	The calling task's job was not created by the Human Interface.
E$CONNECTION$-OPEN	At least one of the following is true: • The connection to :CI: was not open for reading or the connection to :CO: was not open for writing. • The connection to :CI: or :CO: was not open. • The connection to :CI: or :CO: was opened with A$OPEN rather than S$OPEN.
E$EXIST	The token value for :CI: or :CO: is not a token for an existing object.
E$FLUSHING	The device containing the :CI: and :CO: files was being detached.
EIOHARD	While attempting to access the :CI: or :CO: file, the Operating System detected a hard I/O error.
EIOOPRINT	While attempting to access the :CI: or :CO: file, this call found that the device was off-line. Operator intervention is required. C$FORMAT$EXCEPTION returns the value EIONOT$READY for this code.

EIOSOFT	While attempting to access the :CI: or :CO: file, this call detected a soft I/O error. It tried again, but was unsuccessful. Another try might be successful.
EIOUNCLASS	An unknown type of I/O error occurred while this call tried to access the :CI: or :CO: file.
EIOWRPROT	While attempting to obtain a connection to the :CO: file, this call found that the volume containing the file is write-protected.
E$LIMIT	At least one of the following is true: • The calling task's job has already reached its object limit. • The calling task's job, or the job's default user object, is already involved in 255 (decimal) I/O operations. • The calling task's job was not created by the Human Interface.
E$MEM	The memory available to the calling task's job is not sufficient to complete the call.
ENOTCONNECTION	The call obtained a token for an object that should have been a connection to :CI: or :CO: but was not a file connection.
E$PARAM	The call attempted to write beyond the end of a physical file.
E$SPACE	One of the following is true: • The output volume is full. • The call attempted to write beyond the end of a physical file.
E$STREAM$SPECIAL	When attempting to read or write to :CI: or :CO:, the Extended I/O System issued an invalid stream file request.
E$SUPPORT	The connection to :CI: or :CO: was not created by this job.
E$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:

message$p	response$max	Action
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$RESPONSE and C$SENDEORESPONSE 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$OK	No exceptional conditions were encountered.
E$CONNECTION$-OPEN	At least one of the following is true: • The connection to :CI: was not open for reading or the connection to :CO: was not open for writing. • The connection to :CI: or :CO: was not open. • The connection to :CI: or :CO: was opened with A$OPEN rather than S$OPEN.
E$CONTEXT	The calling task's job was not created by the Human Interface.
E$ERROR$-OUTPUT	Attempted to call SENDEORESPONSE through an invalid method.
E$EXIST	The token value for :CI: or :CO: is not a token for an existing object.
E$FLUSHING	The device containing the :CI: and :CO: files was being detached.
EIOOPRINT	While attempting to access the terminal, this call found that the device was off-line. Operator intervention is required. C$FORMAT$EXCEPTION returns the value EIONOT$READY when given this code.

E$LIMIT	At least one of the following is true: • The calling task's job has already reached its object limit. • The calling task's job or the job's default user object is already involved in 255 (decimal) I/O operations. • The calling task's job was not created by the Human Interface.
E$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.
ENOTCONNECTION	The call obtained a token for an object that should have been a connection to :CI: or :CO: but was not a file connection.
E$PARAM	The call attempted to write beyond the end of a physical file.
E$STREAM$SPECIAL	When attempting to read or write to :CI: or :CO:, the Extended I/O System issued an invalid stream file request.
E$SUPPORT	The connection to the terminal was not created by this job.
E$TIME	The calling task's job was not created by the Human Interface.

CSETPARSE$BUFFER

CSETPARSE$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

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

When called, CSETPARSE$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 CGETCHAR.

Note that CSETPARSE$BUFFER does not affect the buffer from which CGETINPUT$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$LIMIT	At least one of the following is true: • The calling task's job has already reached its object limit. • This indicates that the calling task's job was not created by the Human Interface.
E$MEM	The memory available to the calling task's job is not sufficient to complete the call.

***

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.

- 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
EJOBTABLES	8081H
E$DEFAULT$SO	8083H
E$STRING	8084H
E$ERROR$OUTPUT	8085H
Environmental Errors:	
E$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
E$DATE	008CH
ENOPARAMETER	008DH
E$VERSION	008EH
EGETPATH$ORDER	008FH

The values of condition codes fall into ranges based on the iRMX 86 layer which first detects the condition. Table B-2 lists the layers and their respective ranges, with numeric values expressed in hexadecimal notation. Table B-3 lists all the exception codes for the operating system. All the exception codes are listed to their type (environmental errors, Nucleus programming errors, etc.). For more information on the exception codes, consult the manual which describes the layer from which the exception code originates.

Table B-2. Condition Code Ranges

Layer	Environmental Conditions	Programming Errors
Nucleus	0H to 1FH	8000H to 801FH
I/O Systems	20H to 5FH	8020H to 805FH
Application Loader	60H to 7FH	8060H to 807FH
Human Interface	80H to AFH	8080H to 80AFH
Universal Development Interface	C0H to DFH	80C0H to 80DFH
Reserved for Intel *	E0H to 3FFFH	80E0H to BFFFH
Reserved for users	4000H to 7FFFH	C000H to FFFFH

Note: * Exception codes in this range (130 to 14FH; 8130 to 814FH) 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.

Table B-3. Conditions And Their Codes

Category/ Mnemonic	Meaning	Numeric Code	
		Hex	Decimal
E$OK	The most recent system call was successful.	0H	0
Nucleus Environmental Conditions			
E$TIME	A time limit (possibly a limit of zero time) expired without a task's request being satisfied.	1H	1
E$MEM	There is not sufficient memory available to satisfy a task's request.	2H	2
E$BUSY	Another task currently has access to the data protected by a region.	3H	3
E$LIMIT	A task attempted an operation which, if it had been successful, would have violated a Nucleus-enforced limit.	4H	4
E$CONTEXT	A system call was issued out of context or the Operating System was asked to perform an impossible operation.	5H	5
E$EXIST	A token parameter has a value which is not the token of an existing object.	6H	6
E$STATE	A task attempted an operation which would have caused an impossible transition of a task's state.	7H	7
ENOTCONFIGURED	This system call is not part of the present configuration.	8H	8
E$INTERRUPT$SATURATION	An interrupt task has accumulated the maximum allowable number of SIGNAL$INTERRUPT requests.	9H	9
E$INTERRUPT$OVERFLOW	An interrupt task has accumulated more than the maximum allowable amount of SIGNAL$INTERRUPT requests.	0AH	10

Table B-3. Conditions And Their Codes (continued)

Category/ Mnemonic	Meaning	Numeric Code	
		Hex	Decimal
I/O System Environmental Conditions			
E$FEXIST	The specified file already exists.	20H	32
E$FNEXIST	The specified file does not exist.	21H	33
E$DEVFD	The device driver and file driver are incompatible.	22H	34
E$SUPPORT	The combination of parameters entered is not supported.	23H	35
E$EMPTY$-ENTRY	The specified entry in a directory file is empty.	24H	36
EDIREND	The specified directory entry index is beyond the end of the directory file.	25H	37
E$FACCESS	The connection does not have the correct access to the file.	26H	38
E$FTYPE	The requested operation is not valid for this file type.	27H	39
E$SHARE	The requested operation attempted an improper kind of file sharing.	28H	40
E$SPACE	There is no space left on the volume.	29H	41
E$IDDR	An invalid device driver request occurred.	2AH	42
E$IO	An I/O error occurred.	2BH	43
E$FLUSHING	The connection specified in the call was deleted before the operation completed.	2CH	44
E$ILLVOL	The device contains an invalid or improperly-formatted volume.	2DH	45
EDEVOFF-LINE	The device being accessed is now offline.	2EH	46
E$IFDR	An invalid file driver request occurred.	2FH	47

Table B-3. Conditions And Their Codes (continued)

Category/ Mnemonic	Meaning	Numeric Code	
		Hex	Decimal
I/O System Environmental Conditions (continued)			
E$FRAGMENT-ATION	The file is too fragmented to be extended.	30H	48
EDIRNOT$-EMPTY	The call is attempting to delete a directory that is not empty.	31H	49
ENOTFILE$-CONN	The connection parameter is not a file connection, but it should be.	32H	50
ENOTDEV-ICE$CONN	The connection parameter is not a device connection, but it should be.	33H	51
E$CONN$NOT$-OPEN	The connection is either closed or it is open for access not compatible with the current request.	34H	52
E$CONN$OPEN	The task attempted to open a connection that is already open.	35H	53
E$BUFFERED$-CONN	The specified connection was opened by the EIOS, which specified one or more buffers for the connection.	36H	54
E$OUTSTAND-ING$CONNS	A soft detach was specified, but connections to the device still exist.	37H	55
E$ALREADY$-ATTACHED	The specified device is already attached.	38H	56
EDEV-DETACHING	The file specified is on a device that the Operating System is detaching.	39H	57
ENOTSAME$-DEVICE	The existing pathname and the new pathname refer to different devices. You cannot simultaneously rename a file and move it to another device.	3AH	58
E$ILLOGICAL$-RENAME	The call is attempting to rename a directory to a new path containing itself.	3BH	59

Table B-3. Conditions And Their Codes (continued)

Category/ Mnemonic	Meaning	Numeric Code	
		Hex	Decimal
I/O System Environmental Conditions (continued)			
E$STREAM$-SPECIAL	A stream file request is out of context. Either it is a query request and another query request is already queued, or it is a satisfy request and either the request queue is empty or a query request is queued.	3CH	60
E$INVALID$-FNODE	The connection refers to a file with an invalid fnode. You should delete this file.	3DH	61
E$PATHNAME$-SYNTAX	The specified pathname contains invalid characters.	3EH	62
E$FNODE$LIMIT	The volume already contains the maximum number of files. No more fnodes are available for new files.	3F	63
ELOGNAME$-SYNTAX	The specified pathname starts with a colon (:), but it does not contain a second, matching colon.	40H	64
E$IOMEM	The Basic I/O System has insufficient memory to process a request.	42H	66
E$MEDIA	The device containing a specified file is not on-line.	44H	68
ELOGNAME$-NEXIST	The Extended I/O System was unable to find the specified logical name in the object directories that it checks.	45H	69
ENOTOWNER	The user who attempted to detach the device is not the owner of the device.	46H	70
EIOJOB	The Extended I/O System cannot create an I/O job because the size specified for the object directory is too small.	47H	71
EIOUNCLASS	An unknown type of I/O error occurred.	50H	80

Table B-3. Conditions And Their Codes (continued)

Category/ Mnemonic	Meaning	Numeric Code	
		Hex	Decimal
I/O System Environmental Conditions (continued)			
EIOSOFT	A soft I/O error occurred. A retry might be successful.	51H	81
EIOHARD	A hard I/O error occurred. A retry is probably useless.	52H	82
EIOOPRINT	The device was off-line. Operator intervention is required.	53H	83
EIOWRPROT	The volume is write-protected.	54H	84
EIONO$DATA	A tape drive attempted to read the next record, but it found no data	55H	85
EIOMODE	A tape drive attempted a read (write) operation before the previous write (read) completed	56H	86
Application Loader Environmental Conditions			
EBADGROUP	The group definition record contains an invalid group component.	61H	97
EBADHEADER	The object file contains an invalid header record.	62H	98H
EBADSEGDEF	The object file contains an invalid segment definition record.	63H	99H
E$CHECKSUM	A checksum error occurred while reading a record.	64H	100
E$EOF	The Application Loader encountered an unexpected end-of-file while reading a record.	65H	101
E$FIXUP	The file contains an invalid fixup record.	66H	102
ENOLOADER$-MEM	There is insufficient memory to satisfy the memory requirements of the Application Loader.	67H	103

Table B-3. Conditions And Their Codes (continued)

Category/ Mnemonic	Meaning	Numeric Code	
		Hex	Decimal
Application Loader Environmental Conditions (continued)			
ENOMEM	There is insufficient memory to create PIC/LTL segments.	68H	104
ERECFORMAT	The file contains an invalid record format.	69H	105
ERECLENGTH	The record length exceeds the configured size of the Application Loader buffer.	6AH	106
ERECTYPE	The file contains an invalid record type.	6BH	107
ENOSTART	The Application Loader could not find the start address.	6CH	108
EJOBSIZE	The maximum memory-pool size of the job being loaded is smaller than the amount of memory required to load its object file.	6DH	109
E$OVERLAY	The overlay name does not match any of the overlay module names.	6EH	110
E$LOADER$-SUPPORT	The file requires features not supported by the Application Loader as configured.	6FH	111
ESEGBOUNDS	One of the data records in a module loaded by the Application Loader referred to an address outside the segment created for it.	70H	112
Human Interface Environmental Conditions			
E$LITERAL	The parsing buffer contains a literal with no closing quote.	80H	128
E$STRING$BUF-FER	The string to be returned exceeds the size of the buffer the user provided in the call.	81H	129
E$SEPARATOR	The parsing buffer contains a command separator.	82H	130

Table B-3. Conditions And Their Codes (continued)

Category/ Mnemonic	Meaning	Numeric Code	
		Hex	Decimal
Human Interface Environmental Conditions (continued)			
E$CONTINUED	The parse buffer contains a continuation character.	83H	131
E$INVALID$-NUMERIC	A numeric value contains invalid characters.	84H	132
E$LIST	A value in the value list is missing.	85H	133
E$WILDCARD	A wild-card character appears in an invalid context, such as in an intermediate component of a pathname.	86H	134
E$PREPOSITION	The command line contains an invalid preposition.	87H	135
E$PATH	The command line contains an invalid pathname.	88H	136
E$CONTROL$C	The user typed a CONTROL-C to abort the command.	89H	137
E$CONTROL	The command line contains an invalid control.	8AH	138
E$UNMATCHED$-LISTS	The number of files in the input and output pathname lists is not the same.	8BH	139
E$DATE	The operator entered an invalid date.	8CH	140
ENOPARAM-ETERS	A command expected parameters, but the operator didn't supply any.	8DH	141
E$VERSION	The Human Interface is not compatible with the version of the command the operator invoked.	8EH	142
EGETPATH$-ORDER	A command called CGETOUTPUT$PATHNAME before calling CGETINPUT$PATHNAME	8FH	143
UDI Environmental Conditions			
E$UNKNOWN$EXIT	The program exited normally.	0C0H	192

Table B-3. Conditions And Their Codes (continued)

Category/ Mnemonic	Meaning	Numeric Code	
		Hex	Decimal
UDI Environmental Conditions (continued)			
E$WARNING$EXIT	The program issued warning messages.	0C1H	193
E$ERROR$EXIT	The program detected errors.	0C2H	194
E$FATAL$EXIT	A fatal error occurred in the program.	0C3H	195
E$ABORT$EXIT	The Operating System aborted the program.	0C4H	196
EUDIINTERNAL	A UDI internal error occurred.	0C5H	197
Nucleus Programmer Errors			
* E$ZERO$-DIVIDE	A task attempted a divide in which the quotient was larger than 16 bits.	8000H	32768
* E$OVERFLOW	An overflow interrupt occurred.	8001H	32769
E$TYPE	A token parameter referred to an existing object that is not of the required type.	8002H	32770
E$PARAM	A parameter that is neither a token nor an offset has an invalid value.	8004H	32772
EBADCALL	An OS extension received an invalid function code.	8005H	32773
* E$ARRAY$-BOUNDS	Hardware or software has detected an array overflow.	8006H	32774
* ENDPSTATUS	A Numeric Processor Extension (NPX) error has occurred. OS extensions can return the status of the NPX to the exception handler.	8007H	32775
* E$ILLEGAL$-OPCODE	The iAPX 186 or 286 processor tried to execute an invalid instruction	8008H	32776

* For iAPX 286-based systems, a CPU trap caused this exceptional condition.

Table B-3. Conditions And Their Codes (continued)

Category/ Mnemonic	Meaning	Numeric Code Hex	Decimal
Nucleus Programmer Errors (continued)			
* E$EMULATOR$-TRAP	The iAPX 186 or 286 processor tried to execute an ESC instruction with the "emulator" bit set in the relocation register (iAPX 186) or the machine status word (iAPX 286).	8009H	32777
* E$INTERRUPT$-TABLE$LIMIT	An iAPX 286 LIDT instruction changed the interrupt table limit to a value between 20H and 42H.	800AH	32778
* E$CPUXFER$-DATA$LIMIT	For an iAPX 286 processor, the processor extension data transfer exceeded the offset of 0FFFFH in a segment.	800BH	32779
* ESEGWRAP$-AROUND	For an iAPX 286 processor, either a word operation attempted a segment wraparound at offset 0FFFFH; or a PUSH, CALL, or INT instruction attempted to execute while SP = 1.	800CH	32780
E$CHECK$EX-CEPTION	A Pascal task has exceeded the bounds of a CASE statement.	8017H	32791
I/O System Programmer Errors			
E$NOUSER	No default user is defined.	8021H	32801
E$NOPREFIX	No default prefix is defined.	8022H	32802
ENOTLOG$NAME	The specified object is not a device connection or file connection.	8040H	32832
ENOTDEVICE	A token parameter referred to an existing object that is not, but should be, a device connection.	8041H	32833

* For iAPX 286-based systems, a CPU trap caused this exceptional condition.

Table B-3. Conditions And Their Codes (continued)

Category/ Mnemonic	Meaning	Numeric Code	
		Hex	Decimal
I/O System Programmer Errors (continued)			
ENOTCON-NECTION	A token parameter referred to an existing object that is not, but should be, a file connection.	8042H	32834
Application Loader Programmer Error			
EJOBPARAM	The maximum memory pool size specified for the job is less than the minimum memory pool size specified.	8060H	32864
Human Interface Programmer Errors			
E$PARSE$TABLES	There is an error in the internal parse tables.	8080H	32896
EJOBTABLES	An internal Human Interface table was overwritten, causing it to contain an invalid value.	8081H	32897
E$DEFAULT$SO	The default output name string is invalid.	8083H	32898
E$STRING	The pathname to be returned exceeds 255 characters in length.	8084H	32899
E$ERROR$OUTPUT	The command invoked by C$SEND$COMMAND includes a call to C$SEND$EO$RESPONSE, but the command connection does not permit C$SEND$EO$RESPONSE calls.	8085H	32900
UDI Programmer Errors			
E$RESERVE$-PARAM	The calling program tried to reserve memory for more than 12 files or buffers.	80C6H	32966
E$OPEN$PARAM	The calling program requested more than two buffers when opening a file.	80C7H	32967

***

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.

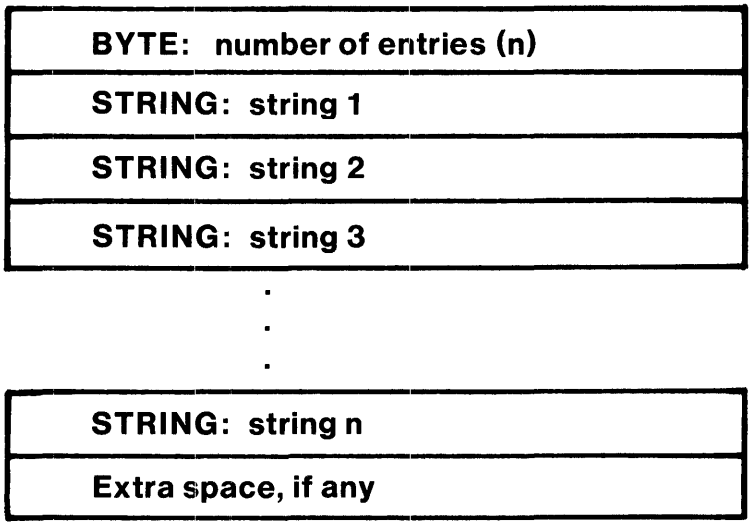

Figure C-1. String Table Format

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

Primary references are underscored.