900 Series HP 3000 Computer Systems

Command Interpreter Access and Variables Programmer's Guide

HP Part No. 32650-90011
Printed in U.S.A. 1994

Fourth Edition
E0494

Hewlett-Packard Company
3000 Hanover Street
Palo Alto, CA 94304 U.S.A.

Printing History

The following table lists the printings of this document, together with the respective release dates for each edition. The software version indicates the version of the software product at the time this document was issued. Many product releases do not require changes to the document. Therefore, do not expect a one-to-one correspondence between product releases and document editions.

Edition	Date	Software Version
First Edition	November 1987	A.01.00
First Edition Update 1	July 1988	A.10.00
Second Edition	April 1990	A.40.00
Third Edition	June 1992	B.40.00
Fourth Edition	April 1994	C.50.00

Preface

MPE/iX, Multiprogramming Executive with Integrated POSIX, is the latest in a series of forward-compatible operating systems for the HP 3000 line of computers.

In HP documentation and in talking with HP 3000 users, you will encounter references to MPE XL, the direct predecessor of MPE/iX. MPE/iX is a superset of MPE XL. All programs written for MPE XL will run without change under MPE/iX. You can continue to use MPE XL system documentation, although it may not refer to features added to the operating system to support POSIX (for example, hierarchical directories).

Finally, you may encounter references to MPE V, which is the operating system for HP 3000s, not based on PA-RISC architecture. MPE V software can be run on the PA-RISC (Series 900) HP 3000s in what is known as *compatibility mode*.

This manual provides a programmer's view of the command interpreter (CI) and its programming capabilities. Through command files and user-defined commands (UDCs), the programmer can develop scripts to accomplish many system-oriented tasks simply and efficiently.

This manual presents these capabilities and examples of their use in the following sequence:

Chapter 1, "Introduction," describes the CI and how it is viewed by all system users. It further describes the unique attributes that make it especially useful to programmers.

Chapter 2, "Accessing the Command Interpreter," presents the various methods of invoking commands. It describes command files and UDCs in detail.

Chapter 3, "Setting and Manipulating Variables," describes user-defined variables and predefined variables. It details the methods of setting and modifying any variable, as well as creating, naming, and deleting user-defined variables.

Chapter 4, "Evaluating Expressions," describes the expression evaluator and its numerous functions. It provides examples of performing arithmetic and bit operations, and evaluating strings and file characteristics.

Chapter 5, "Using Language Constructs Available with the CI," presents key functions that provide variations in receiving input and delivering output. It provides the basis for conditional branching and looping structures.

Chapter 6, "Accessing Variables and CI Commands in Applications," describes the intrinsics that provide communication between application programs and built-in CI commands, command files, and UDCs.

Chapter 7, "Sample Command Files," provides samples of actual command files offer suggested uses of many of the CI functions. A short description of each points out some of the key processing techniques that can be accomplished with the CI.

Chapter 8, "Command Input/Output Redirection (CIOR)," describes how to define different files for command input and command output.

Two appendixes provide lists of commonly used predefined variables and evaluator functions. Detailed explanations of all predefined variables, evaluator functions, and CI commands can be found in the *MPE/iX Commands Reference Manual* (32650-90003).

Conventions

`UPPERCASE` In a syntax statement, commands and keywords are shown in uppercase characters. The characters must be entered in the order shown; however, you can enter the characters in either uppercase or lowercase. For example:

```
COMMAND
```

can be entered as any of the following:

```
command        Command        COMMAND
```

It cannot, however, be entered as:

```
comm           com_mand       comamnd
```

italics In a syntax statement or an example, a word in italics represents a parameter or argument that you must replace with the actual value. In the following example, you must replace *filename* with the name of the file:

`COMMAND` *filename*

bold italics In a syntax statement, a word in bold italics represents a parameter that you must replace with the actual value. In the following example, you must replace filename with the name of the file:

```
COMMAND(filename)
```

punctuation In a syntax statement, punctuation characters (other than brackets, braces, vertical bars, and ellipses) must be entered exactly as shown. In the following example, the parentheses and colon must be entered:

(*filename*):(*filename*)

<u>underlining</u> Within an example that contains interactive dialog, user input and user responses to prompts are indicated by underlining. In the following example, <u>yes</u> is the user's response to the prompt:

```
Do you want to continue? >>  yes
```

`{ }` In a syntax statement, braces enclose required elements. When several elements are stacked within braces, you must select one. In the following example, you must select either `ON` or `OFF`:

`COMMAND` $\left\{ \begin{matrix} \texttt{ON} \\ \texttt{OFF} \end{matrix} \right\}$

`[ ]` In a syntax statement, brackets enclose optional elements. In the following example, `OPTION` can be omitted:

`COMMAND` *filename* `[OPTION]`

When several elements are stacked within brackets, you can select one or none of the elements. In the following example, you can select `OPTION` or *parameter* or neither. The elements cannot be repeated.

`COMMAND` *filename* $\left[\begin{matrix} \texttt{OPTION} \\ \textit{parameter} \end{matrix} \right]$

Conventions (continued)

`[ ... ]`

In a syntax statement, horizontal ellipses enclosed in brackets indicate that you can repeatedly select the element(s) that appear within the immediately preceding pair of brackets or braces. In the example below, you can select *parameter* zero or more times. Each instance of *parameter* must be preceded by a comma:

`[,`*parameter*`][...]`

In the example below, you only use the comma as a delimiter if *parameter* is repeated; no comma is used before the first occurrence of *parameter*:

`[`*parameter*`][,...]`

`| ... |`

In a syntax statement, horizontal ellipses enclosed in vertical bars indicate that you can select more than one element within the immediately preceding pair of brackets or braces. However, each particular element can only be selected once. In the following example, you must select **`A`**, **`AB`**, **`BA`**, or **`B`**. The elements cannot be repeated.

$$\left\{ \begin{matrix} \texttt{A} \\ \texttt{B} \end{matrix} \right\} \texttt{| ... |}$$

`...`

In an example, horizontal or vertical ellipses indicate where portions of an example have been omitted.

`Δ`

In a syntax statement, the space symbol `Δ` shows a required blank. In the following example, *parameter* and *parameter* must be separated with a blank:

`(`*parameter*`)Δ(`*parameter*`)`

The symbol indicates a key on the keyboard. For example, RETURN represents the carriage return key or Shift represents the shift key.

CTRL*character*

CTRL*character* indicates a control character. For example, CTRL**`Y`** means that you press the control key and the Y key simultaneously.

Contents

Figures

Tables

1

Introduction

The user's interface to the operating system (MPE/iX) provides a flexible, productive environment for all users. The command interpreter (CI) is an integral part of this interface.

What Is the Command Interpreter?

The CI is an executable program that acts as an interface between the user and MPE/iX.

It consists of two parts: a centralized scanner/parser that scans a command string for valid syntax, and an interpreter that invokes the appropriate command executor based on command input.

The scanner/parser analyzes the command string for proper syntax. If the syntax is incorrect, it is returned for correction. If correct, the command string is passed to the interpreter portion of the CI.

The interpreter identifies the command as a user-defined command, predefined system command, program file, or command file. It then invokes the appropriate procedure to process the command or to run the specified program file or command file.

The CI and many of its commands operate on HFS (hierarchical file system) file names and directories. In addition, several commands are dedicated to operations on HFS files and directories. Refer to *New Features of MPE/iX: Using The Hierarchical File System* (32650-90351) and to *MPE/iX Commands Reference Manual* (32650-90003).

In addition, a CM (compatibility mode) command (plus its parameters and objects) can consist of 253 characters for commands invoking a relative pathname; or it can consist of as many as 255 characters for commands invoking an absolute path name. The maximum size for native mode commands is 511 characters, less the length of the command and and other parameters.

Note

If a POSIX (HFS) object is directly under root or an MPE group, its name length is reduced to a maximum of 16 characters.

How Is the Command Interpreter Used?

The CI is used by everyone on the system. Entering CI commands is the primary method of communication between the user and the operating system. Every logon invokes the CI.

General users access the operating system through logon and logoff commands. A user obtains file information and runs subsystems through CI commands. The general user's view of the CI is described in the self-paced tutorial *Using the 900 Series HP 3000: Fundamental Skills* (32650-60037).

System administrators and operators rely on the CI in many of their duties. System startups, shutdowns, and updates are driven by CI commands to the operating system. Daily maintenance of files, accounts, groups, and users' capabilities are accomplished with CI commands. The CI commands used most often in operational and administrative tasks are described in the *Performing System Management Tasks* (32650-90004) and the *Performing System Operations Tasks* (32650-90137).

Programmers access the CI to compile, link, and run application programs. They can access system information and routines through the CI and the intrinsic mechanism. The CI also offers a programming environment that accesses system-specific information more efficiently than most application languages. This manual concentrates on the CI features that apply to programmers.

How Programmers Use the CI

The CI contains a set of features that provides the key functions of a programming language:

- Command files and user-defined commands—A mechanism for executing multiple CI commands in a prescribed sequence, similar to a procedure.
- Variables—User-defined and predefined variables that can be accessed by the CI and application programs within the same session.
- Expression evaluator—A command processing phase that resolves all arithmetic, string, and boolean expressions.
- Language constructs—Command structures that simulate the branching, looping, input, and output structures needed for programmatic control.
- CI intrinsics—An interface mechanism providing programmatic access to the CI from applications programs.

The programming capability of the CI can be used to simplify tasks. Complicated routines requiring multiple commands can be made transparent to users through the use of command files and user-defined commands (UDCs).

Some programming tasks can be coded more simply and efficiently with CI commands than with a standard application language. Routines can be written with CI commands and accessed from application programs through the intrinsic facility. Variables can be used by both CI routines and application programs to pass information between routines.

2

Accessing the Command Interpreter

Command images are routed to the CI through several different paths. You can issue CI commands in several ways:

- From a job or session.
- Through the `REDO` and `DO` commands.
- Through user-defined commands (UDCs) and command files.
- From application programs using CI intrinsics.
- From the `INFO=` string when a second level of the CI is run.
- From an input file when a second level of the CI is run with its `$STDIN` file redirected.

Issuing Commands Directly

Most often, commands are issued directly from a job or session. The CI is invoked automatically for each interactive session. Its prompt indicates that it is active and awaiting command input. Once a command is entered, the CI parses and executes it, displays requested output, and prompts for another command.

The CI is also activated for each active job stream. CI commands contained in a job stream must be preceded by a "pseudo" CI prompt, usually an exclamation point (!). When the job is streamed, commands preceded by the prompt are identified as input to the CI. The CI verifies the command syntax and executes it. Processing returns to the jobstream for the next executable statement.

Refer to the *MPE/iX Commands Reference Manual* (32650-90003) for a detailed explanation of each command, its purpose, and its parameters.

Reissuing and Modifying Commands

By using the DO and REDO commands, you can modify and reissue a command that has been entered interactively from a session. These commands are especially helpful to correct typing errors or to avoid retyping complicated command input.

The command history stack retains the latest commands that have been issued from the session or job. Usually, the last 20 commands that have been issued are kept in the stack.

The LISTREDO command displays the command history stack. A command can be selected by referring to its position in the stack. Elements of the command image can be changed or deleted as needed. The command can then be reissued.

There are several ways of selecting a command from the stack. Refer to the self-paced tutorial *Using the 900 Series HP 3000: Fundamental Skills* (32650-60037) for details regarding the REDO, LISTREDO, and DO commands and examples of their use.

Issuing Commands Through UDCs

A user-defined command (UDC) saves time and reduces errors. It allows a commonly used command string or multiple command strings to be issued with a single entry.

Creating a UDC File

A UDC file is a specially identified file containing one or more UDCs. Each UDC is composed of a unique name, required and optional parameter specifications, selected options controlling its execution, and one or more command lines. In a UDC file, each UDC is separated from the next by a line of one or more asterisks beginning in column 1. You can use any editor to create a UDC file. The following example shows multiple UDCs in a single file.

```
SJ
OPTION LOGON
SHOWJOB
*****
LF
LISTFILE
*****
ABXYZ A,B=B
OPTION LIST
RUN !A.PUB.SYS
RUN !B.PUB.SYS
RUN XYZ
*****
CF1
CMDFL1
*****
```

To distinguish a UDC file from other files, it must be cataloged, using the SETCATALOG command. This process identifies the contents of the file as UDCs, so that any UDC name within the file is interpreted as a command by the CI. UDC files can be created at the user level, the account level, or the system level. UDCs can be added to or deleted from an existing UDC file, but must be uncataloged first. (Refer to the self-paced tutorial *Using the 900 Series HP 3000: Fundamental Skills* (32650-60037) for details about cataloging, adding to, and deleting from a UDC file.)

Executing a UDC

The UDC file in the preceding example contains four UDCs. The user can enter a UDC name and parameters, if appropriate, at the CI prompt to execute its associated routine. For example, by entering LF at the CI prompt, the UDC named LF is executed. Its associated commands execute the LISTFILE command.

When executing a command, the CI first searches cataloged UDC files for a UDC name that matches the command string. UDC files are searched in the following sequence:

1. User-level UDCs.
2. Account-level UDCs.
3. System-level UDCs.

If the specified command does not match a UDC name, the built-in MPE/iX commands are searched. UDCs, therefore, provide a method of superceding standard MPE/iX commands.

Note

When a UDC or command file is used to invoke a program while in BREAK from another program invoked from within a UDC or command file, MPE/iX returns CIERR 9065.

Specifying UDC Options

Several options are available to control UDC processing. These options are especially useful to programmers to allow or prohibit certain features. (Refer to the self-paced tutorial *Using the 900 Series HP 3000: Fundamental Skills* (32650-60037) for a detailed description of each UDC option.)

LIST or NOLIST	Controls the display of each command image as it is executed.
HELP or NOHELP	Controls the ability to display the contents of the UDC in help mode.
BREAK or NOBREAK	Controls the ability to stop execution of a UDC.
LOGON or NOLOGON	Controls the automatic execution of a UDC at logon.
RECURSION or NORECURSION	Controls the search for UDCs called from another UDC.

`PROGRAM` or `NOPROGRAM` Controls the ability of a UDC to be executed from a program.

The contents of a UDC can be listed as each command is executed using the `LIST` option. All or any portion of the UDC can be kept from listing by specifying the `NOLIST` option. The `NOLIST` option is often used to eliminate unnecessary display or to maintain security. The `NOHELP` option similarly limits the listing of a UDC's contents within the Help facility.

The `NOBREAK` option limits the ability of the user to break the processing of UDC commands. This option is often used to protect a series of processing steps or to ensure that an error procedure is completed. The `NOBREAK` option is also used as a security measure by keeping the user from gaining control of the CI by breaking a procedure.

UDCs can be invoked automatically when the user logs on to the system if the `LOGON` option is set. This feature can be used to restrict users to a particular application environment or to automatically perform a routine commonly performed at the beginning of a session. In the previous example, the first UDC (SJ) contains the logon option. This UDC is performed automatically when you log on. Your logon is followed immediately by the list of all active jobs, the output of the `SHOWJOB` command. The CI prompt is displayed following the job display. Note that only one UDC in the UDC file can contain the `LOGON` option.

The search for a UDC starts at the beginning of the search sequence to the point where a matching UDC name is encountered. If one UDC is called from another, the search for the second UDC begins where the previous search ended. The `RECURSION` option provides a method of specifying that the search must start at the beginning of the standard search sequence.

The `NOPROGRAM` option controls UDC calls from an application program. A UDC can be executed through the `HPCICOMMAND` intrinsic if the `PROGRAM` option is specified in the UDC. This option is often used to control execution of UDCs and built-in commands that have the same name.

Issuing Commands Through Command Files

Command files, like UDCs, can be created to execute single or multiple commands. Unlike UDCs, only one routine can be included in a command file.

Creating a Command File

Any editor can be used to create a command file and to modify or delete a portion of it. The file name is used to invoke the command file. There is no cataloging procedure for a command file.

Command files are often used to test a new user command before establishing it as a UDC. Because they are easier to create and modify than UDCs, command files are also used to execute user commands that change frequently. Command files are not as secure as UDC files because they can be deleted inadvertently with the `PURGE` command (unless it is protected with a lockword, program security, or access rights).

The `PURGE` command permits you to delete files using wildcard specifications, with levels of confirmation. Refer to *MPE/iX Commands Reference Manual* (32650-90003).

For information on HFS (hierarchical file system) security provisions, refer to *New Features of MPE/iX: Using The Hierarchical File System* (32650-90351).

Executing a Command File

To execute a command file, enter the command file name at the CI prompt. In processing any command, the CI first checks the UDC files and MPE/iX built-in commands. If no match is found, the CI automatically searches for a program file or command file of the same name. The CI uses the following sequence to identify the command input it has received:

1. User-, account-, and system-level UDCs.
2. Built-in MPE/iX commands.
3. Program and command files.

If no group or account is specified for the command file name, a search pathway determines the group and account sequence for the search. By default, the current group is checked first, followed by the `PUB` group of the logon account, followed by `PUB.SYS`. This search pathway can be altered by modifying the predefined variable `HPPATH`. (Modifying predefined variables is described in the following chapter.)

Note

To change the `HPPATH` variable to contain your current working directory—in HFS (hierarchical file system) syntax—execute this command:

```
SETVAR HPPATH "!!HPCWD:"
```

Specifying Options for Command Files

The RECURSION and LOGON options specified for UDCs are not applicable to command files. Other options available with UDCs can be used in command files.

If a command, UDC, or another command file is called from a command file, the standard search path is used: UDCs, built-in MPE/iX commands, command or program files. The RECURSION option, therefore, is unnecessary for command files.

Command files cannot be invoked automatically when a user logs on to the system. A command file can be executed at logon, however, by calling the command file through a logon UDC.

Issuing Commands from an Application Program

The intrinsic mechanism allows applications to call system-defined functions. The mechanism is implemented differently for each application language. (Refer to the appropriate programming language user's manual for details.)

The CI can be invoked from an application program by using the HPCICOMMAND or COMMAND intrinsics. The HPCICOMMAND intrinsic activates the CI to execute the command identified in the intrinsic's parameters. Any built-in MPE command, UDC, or command file can be executed this way. When completed, control is returned to the calling program. The COMMAND intrinsic works in a similar manner, but only built-in MPE/iX commands can be executed.

Parameter Handling

The rules for the user command header (PARM line and OPTIONs) have been relaxed, command files can now be variable record width files (UDCs still cannot), and a new parameter type, ANYPARM, supports syntax-free user commands.

The reserved words PARM, ANYPARM and OPTION begin statements which constitute the user command header. For UDCs only, parameters may be defined in the same line as the UDC name, with additional parameters described in subsequent PARM or ANYPARM lines. Parameter and option lines may be interspersed. There is a limit of 255 parameters per user command. If ANYPARM is specified it must be the last parameter definition statement, meaning that PARM cannot follow an ANYPARM declaration. The user command header is terminated by the first non-PARM, non-ANYPARM, non-OPTION record. For example:

```
OPTION logon PARM
flag=" doit
```

```
PARM p1, p2=hi PARM
p3=there OPTION
nohelp doit
```

```
OPTION list PARM p1
PARM p2="what's up,
doc?" ANYPARM
p3=what's up, doc?
```

ANYPARM ANYPARM causes all normal delimiters to be ignored. That is,the meaning and delimiters are now considered part of the parameter's value. This definition forces ANYPARM to be the last parameter declared, since the equal sign (=), comma (,), semicolon (;), quotes ("), etc. are treated as part of the ANYPARM parameter's value.

A benefit of ANYPARM can be seen in the following example:

```
MYTELL user, word1=",w2=",w3=",w4=",w5=",w6=",w7=",w8=",w9=",w10="
tell !user; !hpdatef (!hptimef) -- !word1 !w2 !w3 !w4 !w5 !w6 !w7 !w8 !w9 !w10
***

:mytell zinta.ui This will work; however, we may need some better examples?
FROM/S505 JEFF.UI/MON, OCT 30, 1989 ( 3:11 PM) -- This will work however we...
```

Note A longer message requires quoting, and the ";" and "," are lost since they are delimiters.

Then with the use of ANYPARM.

```
MYTELL user
ANYPARM message=<no message>
tell !user; !hpdatef (!hptimef)-- !message
***

:mytell mikep.uis Will woff;tr,i,d;lev 0,1;mr sp sp-40;mr pc r2 work??
FROM/S505 JEFF.UI/MON, OCT 30, 1989 ( 3:11 PM) -- Will woff;tr,i,d;lev 0,1;...
```

The first example is limited to 10 words, unless quoting is used. The ANYPARM version does not require the user to quote or count parameters, and all normal delimiters are ignored, and thus treated as data.

3

Setting and Manipulating Variables

The CI provides a method of setting, displaying, and deleting variables for each session. Variables are defined by the numeric, string, or boolean values assigned to them. Job control words (JCWs), 16-bit numeric variables, are a subset of MPE/iX variables.

Variables are maintained in the session's variable table, a list of currently defined variables and their values that is established and maintained for each session or job. A variable can be accessed by any command issued from the session. Predefined variables are supplied by the CI to access system information easily from a session or job. You can also establish user-defined variables as needed for your session or job.

Setting User-Defined Variables

Variables are easily set using the `SETVAR` command. The command parameters are the variable name and its value. The value can be specified as a single numeric value, a character string, a boolean value, an expression, or the value of another variable. The `SHOWVAR` command displays the current value of a variable.

In the following example, several variables are set to numeric, string, and boolean values. The `SHOWVAR` command is used to display the current value of specified variables. Note that wildcard characters, such as the at sign (@), question mark (?), and pound sign (#), can be used to display multiple variables that have similar names.

```
:SETVAR CM_PAY_AMT 1000
:SETVAR CM_FIRST_NAME "CAROL"
:SETVAR CM_LAST_NAME "SMITH"
:SETVAR CM_DONE FALSE
:SHOWVAR CM_PAY_AMT,@NAME
CM_PAY_AMT = 1000
CM_FIRST_NAME = CAROL
CM_LAST_NAME = SMITH
```

Defining Variable Type

The variable type is defined by your input. Quotation marks specify that the enclosed phrase is a character string. In the previous example, the first and last names are interpreted as string variables. The value 1000, an unquoted numeric string, is interpreted as an integer value. The unquoted word, FALSE, sets the variable to a boolean value. Expressions can also be used to set variable values.

An unquoted string is interpreted as a variable name, not a character string. In such cases, the first variable is loaded with the value of the second variable. The content of the second variable determines the variable type of the new one. In the following example, one new variable `CM_NAME` is set to the current value of `CM_FIRST_NAME`, which was set in a previous example.

```
:SETVAR CM_NAME CM_FIRST_NAME
:SHOWVAR CM_NAME
CM_NAME = CAROL
```

Displaying User-Defined Variables

As shown in the previous examples, the `SHOWVAR` command displays the current value of specified user-defined variables. If no parameter information is provided, all user-defined variables for the session are displayed in the order they were created.

```
:SHOWVAR
CM_DONE = FALSE
CM_FIRST_NAME = CAROL
CM_LAST_NAME = SMITH
CM_PAY_AMT = 1000
CM_NAME = CAROL
```

Naming Variables

Variables remain set for the duration of the session unless they are deleted or reset to a new value. To avoid collisions with variable names used in command files and UDCs, develop a naming convention that creates unique variable names.

The naming convention used in the preceding examples adds a common prefix to all variables in the command file. In this case, all variables are preceded by `CM_`. Such a prefix could be the name or abbreviated name of the command file itself. This would provide immediate recognition of the command file that set the variable.

Deleting User-Defined Variables

The `DELETEVAR` command deletes user-defined variables. It is easier to delete variables in a single maintenance step if each variable name is preceded by a command file identifier. The following example shows how all user-defined variables in the CM command file are deleted by specifying the command file identifier and a wildcard character.

```
:DELETEVAR CM_@
```

Using Predefined Variables

The CI provides predefined variables, giving the user access to system information. The names and default values are preset in the session variable table.

The following list categorizes the type of system information available and some of the predefined variables that access it. (A listing of commonly used predefined variables is provided in Appendix A. Refer to the *MPE/iX Commands Reference Manual* (32650-90003) for a listing of all predefined variables available.)

Directory information	HPACCOUNT, HPGROUP, HPUSER, HPCWD
User's capabilities	HPUSERCAP, HPUSERCAPF, HPACCTCAP, HPGROUPCAP
User's environment	HPREDOSIZE, HPPATH, HPPROMPT, HPERRDUMP, HPTYPEAHEAD, HPPIN, HPFILE
Input/output specifications	HPLDEVIN, HPINTERACTIVE, HPSTDLIST
Error handling information	CIERROR, JCW, HPCIERRMSG, HPMSGFENCE, HPCIERR, HPFSERR
Job information	HPJOBFENCE, HPSESLIMIT, HPJOBTYPE, HPJOBNAME
System configuration information	HPCONSOLE, HPCPUNAME, HPSYSNAME
System date and time	HPDATE, HPDATEF, HPTIMEF, HPYEAR

Displaying Values

A list of all variables and their current values can be displayed with the SHOWVAR command followed by the wildcard character @. Both predefined variables and any user-defined variables in the session table are displayed.

```
:SHOWVAR @
```

As with user-defined variables, the current value of specific predefined variables can be displayed using the SHOWVAR command and the variable name. In the following example, the HPREDOSIZE, HPPATH, HPCWD, and HPPROMPT variables are displayed.

```
:SHOWVAR HPPROMPT
HPPROMPT = :


:SHOWVAR HPPATH, HPREDOSIZE
HPPATH = !HPGROUP,PUB,PUB.SYS
HPREDOSIZE = 20

:SHOWVAR HPCWD
HPCWD = /SK72NM/PUB/somehfsdir
```

In this last example, `/SK72NM/PUB` is an /ACCOUNT/GROUP path (in hierarchical file system) that corresponds to an MPE/iX GROUP.ACCOUNT, and `somehfsdir` is a directory (below `PUB`).

Modifying Values

No predefined variables can be deleted by the user, however, some of them are modifiable. (Refer to the *MPE/iX Commands Reference Manual* (32650-90003) for a list of predefined variables that cannot be modified.) You can change the default setting of any modifiable predefined variable to better suit your needs. The following example alters the CI prompt and the size of the command history stack using the `SETVAR` command.

```
:SHOWVAR HPPROMPT
HPPROMPT = :
:SETVAR HPPROMPT "==>"
==>


:SHOWVAR HPREDOSIZE
HPREDOSIZE = 20
:SETVAR HPREDOSIZE 25
:SHOWVAR HPREDOSIZE
HPREDOSIZE = 25
```

Accessing Variable Values

The value of a variable can be accessed by dereferencing the variable by name. This substitutes the value of the variable for the variable name. Variables can be dereferenced in three ways: implicitly, explicitly, and recursively.

Implicit Dereferencing

Implicit dereferencing is simply the substitution of the variable's value for the variable name. Only certain commands use implicit dereferencing. Variables used in expressions in the `IF`, `WHILE`, `SETVAR`, and `CALC` commands are dereferenced implicitly.

In these commands, any non-numeric or unquoted string in any expression is assumed to be a variable name. (Exceptions to this are the boolean values TRUE and FALSE, and other special cases, such as WARN, FATAL, and SYSTEM.) In the following example, the variable `CM_NAME` is set implicitly with the contents of the variable `CM_FIRST_NAME`, set in a previous example.

```
:SETVAR CM_NAME CM_FIRST_NAME
:SHOWVAR CM_NAME
CM_NAME = CAROL
```

Explicit Dereferencing

Any variable can be dereferenced explicitly by preceding the variable name with an exclamation point. Explicit dereferencing is available with every CI command. The variable's value rather than its name is substituted in the statement or expression. Explicit dereferencing is often used to include a variable value in an expression.

```
:SETVAR X "Blue"
:SETVAR Y "!X is the best color."
:SHOWVAR X,Y
X = Blue
Y = Blue is the best color.
```

In the preceding example, the value of variable `X` is substituted when the value of variable `Y` is set. The word "Blue" is inserted in the expression where the variable name is dereferenced.

Recursive Dereferencing

Recursive dereferencing is used to set a variable with the dynamic value of another variable. The existing variable's value is not loaded into the new variable until the new variable itself is explicitly dereferenced, guaranteeing that the current value is inserted. Recursive dereferencing is valid only when explicit dereferencing is being used. To specify recursive dereferencing, precede the variable name with two or more exclamation points.

In the following example, the `ECHO` command demonstrates how a variable is dereferenced. `ECHO` requires explicit dereferencing to display the contents of a variable. The `SHOWVAR` command displays the variable name and the value it contains. Using the variables `X` and `Y` set in the previous example, notice how recursive dereferencing provides the most current value.

```
:SETVAR Y "!X is the best color."
:SETVAR Z "!!X is best."
:SHOWVAR Y
Y = Blue is the best color.
:SHOWVAR Z
Z = !X is best.
:ECHO !Y
Blue is the best color.
:ECHO !Z
Blue is best.
:SETVAR X "Red"
:SHOWVAR Y
Y = Blue is the best color.
:SHOWVAR Z
Z = !X is best.
:ECHO !Y
Blue is the best color.
:ECHO !Z
Red is best.
```

Note that X was not dereferenced fully when the variable Z was set because recursive dereferencing of the variable was specified. The value of X was substituted only when the ECHO command was executed, requiring the dereferencing of Z itself.

For the variable Y, however, X was dereferenced when its value was loaded to Y. The value of Y always remained the same and did not reflect any change when X was reset to a new value.

The predefined variable HPPATH is the default search path used to locate UDCs and command files when file names are not fully qualified. By default, it is composed of the current group, the current account's PUB group, and the PUB.SYS group. The current group is identified as the dereferenced value of the variable, HPGROUP. The following example shows that by using recursive dereferencing, the value of HPGROUP remains current.

```
:SHOWVAR HPPATH
HPPATH = !HPGROUP,PUB,PUB.SYS
:SETVAR HPPATH "!!HPGROUP,SCRIPTS.SYS,PUB.SYS"
:SHOWVAR HPPATH
HPPATH = !HPGROUP,SCRIPTS.SYS,PUB.SYS
```

If the user changes groups, the search path automatically adjusts to the current location. The variable value is not substituted until the variable is explicitly dereferenced at command execution.

Substituting Strings

Variables are dereferenced from left to right. Their values are substituted by the CI before the command is parsed or executed. String variables and constants can be concatenated to create a new string. Variables and strings can be joined to create a new variable value.

To concatenate a string variable with a prefix, place the prefix immediately before the exclamation point for the variable. If spacing is desired, the appropriate number of blanks must be inserted between the two values.

In the following example, the ECHO command is used to display a string made up of a string literal and the value of a variable that was set in a previous example. Note that the ECHO command requires explicit dereferencing to display the contents of a variable.

```
:ECHO GRADUATE: !CM_LAST_NAME
GRADUATE: SMITH
```

If a dereferenced variable is enclosed in quotation marks, any dereferencing within the quotation marks is performed first. The following example demonstrates the difference between string substitution sequences.

```
:SETVAR VAR1 "ABC"
:SETVAR VAR 'hello'
:SETVAR A 1
:ECHO !VAR!A
hello1
:ECHO !"VAR!A"
ABC
```

The first `ECHO` command is dereferenced from left to right. The value of the variable `VAR` (hello) is concatenated with the value of the variable `A` (1). No space was inserted between the two variables.

Since `VAR!A` in the second `ECHO` command is enclosed in quotes, the `A` is dereferenced first. The result, the variable name `VAR1`, is then dereferenced to its value ABC.

Variables can be loaded with a string value composed of other variables and strings. It is important to remember that the type of variable is defined by the data loaded into it. String data must be enclosed in quotes to define its type. In this way, explicit or recursive dereferencing must be performed to differentiate between variable names and strings. The following example creates a single variable for a name by combining the contents of the first and last name variables.

```
SETVAR CM_NAME  "!CM_FIRST_NAME !CM_LAST_NAME"
SHOWVAR CM_NAME
CM_NAME = CAROL SMITH
```

Expressions can also be used to set a variable value. The following example uses an algebraic equation to create the same result as in the previous example.

```
:SETVAR CM_NAME  CM_FIRST_NAME + ‘‘ ’’ + CM_LAST_NAME
SHOWVAR CM_NAME
CM_NAME = CAROL SMITH
```

Note that the string “ ” was added to the equation to maintain the proper spacing. Without this, adding the two variables would have concatenated the variable values.

4

Evaluating Expressions

Expression results can be used to define complex functions. They form the basis of comparison structures in establishing branches and loops. Expressions can also be used to set variables based on an equation of predefined variables, user-defined variables, constants, and arithmetic or logical operators.

Using Expressions in CI Commands

The CI provides an expression evaluator that supports a large selection of arithmetic operations; number conversions; file information; and string, bit, and variable operations. (Appendix B provides a table of commonly used functions. Refer to the *MPE/iX Commands Reference Manual* (32650-90003) for a complete list of expression evaluator functions.)

Performing Arithmetic Operations

Standard addition, subtraction, multiplication, and division functions, as well as exponentiation, absolute value, and modulo arithmetic functions, are valid arithmetic operations in the CI. The standard expression symbols are used in the CI as in other programming languages.

The following examples show the common arithmetic symbols used by the CI.

```
Description              Example       Result

Addition                   1 + 3          4
Subtraction                3 - 1          2
Multiplication             2 * 4          8
Division                  16 / 8          2
Absolute Value          ABS(-3)          3
Exponentiation              2^3          8
Multiple Functions    2 * 3 + 6         12
```

The expression evaluator performs integer algebra. Functions are executed from left to right, in the following order of operations.

1. Exponentiation and absolute value.
2. Multiplication and division.
3. Addition and subtraction.

The following example sets the variable EXP with the result of an expression. Note the order in which the operations are performed.

```
:SETVAR A 1
:SETVAR B 2
:SETVAR C 3
:SETVAR D 4
:SETVAR EXP A+B*C/D
:SHOWVAR EXP
EXP = 2
```

The multiplication and division functions are performed first from left to right. The integer result is added to A. Note that division is an integer function; results are truncated, not rounded.

Parentheses can be inserted to clarify or to alter the logical order of operations. The following example provides the expression from the previous example with parentheses included for clarification.

```
:SETVAR EXP A+((B*C)/D)
```

Evaluating Strings

String functions provide a means of analyzing strings. These functions include adding and subtracting characters in a string, determining the length of a string, determining the position of a particular character in a string, adjusting the case of characters, repeating a a character in a string, and removing leading or trailing characters from a string.

Strings are evaluated character by character from left to right. Adding two strings concatenates the contents of the second string to the end of the first.

```
:SETVAR ANS "abcd" + "efg"
:SHOWVAR ANS
ANS = abcdefg
```

Subtracting characters from a string deletes the first occurrence of the specified characters from the original string.

```
:SETVAR ANS "abcabc" - "ab"
:SHOWVAR ANS
ANS = cabc
```

The case of string data can be altered with the downshift (DWNS) or upshift (UPS) functions. These functions are used in case-sensitive comparisons.

```
:SETVAR ANS UPS("abcdEFG")
:SHOWVAR ANS
ANS = ABCDEFG

:SETVAR ANS DWNS("abcdEFG")
:SHOWVAR ANS
ANS = abcdefg
```

The length of a string can be determined with the LEN function.

```
:SETVAR ANS LEN("abcdefg")
:SHOWVAR ANS
ANS = 7
```

The RPT function repeats a string a specified number of times. This is often used to pad another string with blanks or zeros.

```
:SETVAR ANS RPT("A",3)
:SHOWVAR ANS
ANS = AAA
```

The LTRIM and RTRIM functions trim leading or trailing characters from the left or right of a string. Blanks are trimmed from the string unless a trim character is specified in the command.

```
:SETVAR ANS RTRIM("ABC    ") + LTRIM(".....DEF",".")
:SHOWVAR ANS
ANS = ABCDEF
```

ASCII characters can be defined by their ordinal representation. Note that if the parameter of the ordinal function consists of more than one character, the ordinal representation of only the first character is returned. The other characters are ignored.

```
:SETVAR ANS ORD("AbcD")
:SHOWVAR ANS
ANS = 65
```

Ordinal numbers can likewise be defined by their ASCII character representation. This is often used for escape sequences and control sequences that have no printable counterpart.

```
:SETVAR ANS CHR(65)
:SHOWVAR ANS
ANS = A
```

Performing Bit Operations

Integer data can be evaluated and adjusted by using the bit operations provided with the CI. These include bit shift left (LSL), bit shift right (LSR), and circular shifts (CSL and CSR). Also, logical OR, AND, NOT, and XOR operations are available.

Because AM capability is identified by a 1 in the second bit of the predefined variable HPUSERCAP, either of the following expressions can be used to see if a user has AM capability.

```
IF ((HPUSERCAP LSR 30) BAND 1) = 1 THEN ...
```

or

```
IF ODD(HPUSERCAP LSR 30) THEN ...
```

Note that the preceding examples show the use of the bit operations. A simpler way of determining if a user has AM capability would be the following method.

```
IF POS("AM",HPUSERCAPF) > 0 THEN
⋮
```

Converting Numbers

Numeric values can be converted from decimal representation to octal or hexadecimal string equivalents. Numeric output is identified by three prefixes: # represents decimal, $ represents hexadecimal, and % represents octal. If no prefix is specified, decimal is assumed.

In the following example, the decimal number 329 is converted to its hexadecimal and octal string representations.

```
SETVAR ANS HEX(329)
SHOWVAR ANS
ANS = $149

SETVAR ANS OCTAL(329)
SHOWVAR ANS
ANS = %511
```

Note

The results of converting numbers using the `HEX` and `OCTAL` functions are character strings. The result cannot be used in numeric calculations.

Evaluating File Characteristics

The FINFO function retrieves file specifications for the identified file. This is useful in creating files, verifying that a file exists, or identifying file specifications.

The type of file information requested is identified by a numeric entry or an alias for the numeric entry, similar to the parameters of the FINFO function. Note that aliases are strings and must be enclosed in parentheses. String, boolean, or numeric data is returned depending on the option specified.

FINFO recognizes HFS (hierarchical file system) names and supports the FLABELINFO intrinsic items relating to POSIX. Examples include:

- number of hard links
- time of last file access
- file owner (full user.account name and numeric ID)
- file type (directory, symbolic link, pipe, FIFO, device link, etc.)
- record type (fixed, root, spool, byte stream, directory, etc.)
- file size in bytes

- KSAM XL version
- device type (disk, tape, port, streams, sockets, etc.)
- whether the file has been released.

Refer to *New Features of MPE/iX: Using The Hierarchical File System* (32650-90351).

The following table provides some of the most commonly used `FINFO` options, by number and alias, and a description of the results. `FINFO` passes the name of the file using either MPE syntax (the default) or HFS syntax. You can enter CALC `FINFO('.1',0)` and receive TRUE as output, and filename in HFS can end in any character.

If MPE syntax, the file name can include password, group, and account specifications. The file name can backreference a file equation and optionally be preceded by an asterisk.

If HFS syntax, the file name must start with either a dot (.) or a slash (/). For files located in HFS directories, traverse directory entries (TD) access is required to all directories specified in the formal design. If there is no TD access, `FINFO` fails.

If the file can be named using both MPE syntax and HFS syntax (for example, `FILEA.MYGROUP.MYACCT` and `/MYACCT/MYGROUP/FILEA`), the file can be either permanent or temporary. If a temporary and a permanent file have the same name, `FINFO` returns information about the temporary file only. Refer to the *MPE/iX Commands Reference Manual* (32650-90003) for a complete description of the FINFO options.

Table 4-1. FINFO Specifications

Number	Alias	Data Type	Item Description
0	EXIST	Boolean	Existence of file
1	FILENAME ONLY FNAME FULL FILENAME FULLFNAME FULLY QUALIFIED FILENAME	String	File name
2	GROUP GROUPNAME	String	Group name
3	ACCOUNT ACCT ACCOUNTNAME	String	Account name
4	CREATOR	String	File creator name
5	SECURITY MATRIX	String	Security matrix for access
-5	SECURITY MATRIX	Integer	Security matrix for access
6	CREATED CREATION DATE FMTCREATED	String	File creation date
-6	CREATION DATE INTEGER INTCREATED	Integer	File creation date
7	ACCESSED FMTACCESSED LAST ACCESS DATE	String	Last access date
-7	LAST ACCESS DATE INTEGER INTACCESSED	Integer	Last access date
8	MODIFIED LAST MOD DATE FMTMODDATE	String	Last modification date
-8	LAST MOD DATE INTEGER INTMODDATE	Integer	Last modification date
9	FILE CODE MNEMONIC FMTFCODE	String	File code of disk file
-9	FCODE INTFCODE	Integer	File code of disk file
10	USER LABELS WRITTEN	Integer	Number of user labels written
11	USER LABELS AVAIL	Integer	Number of user labels available
12	FILE LIMIT LIMIT	Integer	Total number of logical records possible in the file

Table 4-1. FINFO Specifications (continued)

Number	Alias	Data Type	Item Description
13	FOPTIONS FMTFOPT	String	File options
-13	FORMATTED FOPTIONS INTFOPT	Integer	File options
14	RECORD SIZE RECSIZE	Integer	Record size
15	BLOCK SIZE BLKSIZE	Integer	Block size
16	MAX EXTENTS MAXEXT	Integer	Maximum number of extents
17	LAST EXTENT SIZE LASTEXTSIZE	Integer	Last extent size
18	EXTENT SIZE EXTSIZE	Integer	Extent size
19	END OF FILE EOF	Integer	Number of logical records in file
20	ALLOC TIME FMTALLOCTIME	String	File allocation time
-20	ALLOC TIME INTEGER INTALLOCTIME	Integer	File allocation time
21	ALLOC DATE FMTALLOCDATE ALLOCATED	String	File allocation date
-21	ALLOC DATE INTEGER INTALLOCDATE	Integer	File allocation date
22	NUM OPEN CLOSE RECS	Integer	Number of open/close records
23	DEVICE NAME DEV NAME	String	Device name (8 bytes
24	FMTMODTIME LAST MOD TIME	String	Last modification time
-24	INTMODTIME LAST MOD TIME	Integer	Last modification time
25	FIRST USER LABEL	String	First user label (user label 0)
27	UNIQUE FILE ID UFID	REC	Unique file identifier (UFID)
28	BYPE FILE SIZE BYTEFILESIZE	Integer	Total number of bytes allowed in file

Table 4-1. FINFO Specifications (continued)

Number	Alias	Data Type	Item Description
29	BYTE DATA OFFSET DATASTART	Integer	Start of file offset
30	BYTE RECORD SIZE BYTERECSIZE	Integer	Record size (indicates bytes)
31	BYTE BLOCK SIZE BYTEBLKSIZE	Integer	Block size (indicates bytes)
32	BYTE EXTENT SIZE BYTEEXTSIZE	Integer	Extent size (indicates bytes)
33	LOCKWORD	String	File lockword
34	VOLUME RESTRICTION VOLRESTR	String	Volume restriction
35	VOLUME SET NAME	String	Volume set names
36	LOGSET ID	String	Transaction management log set id
37	LDEV LOGICAL DEVICE NUMBER	Integer	Logical device number
38	POSIX FULL FILE NAME POSIXFULLFNAME	REC	Terminated HFS-syntax system absolute pathname
39	NUM HARD LINKS NUMHARDLINKS	Integer	The current number of hard links to the file
40	ACCESS TIME FMTACCESSTIME LAST ACCESS TIME	String	Time of last file access (clock format)
-40	LAST ACCESS TIME INTEGER INTACCESSTIME	Integer	Time of last file access (clock format)
41	STATUS CHANGE TIME FMTSTATUSCHANGETIME	String	Time of last file status change (clock format)
-41	INTSTATUSCHANGETIME	Integer	Change Time Integer
42	STATUS CHANGE DATE FMTSTATUSCHANGEDATE	String	Date of the last file status change (calendar format)
-42	CHANGE DATE INTEGER INTSTATUSCHANGEDATE	Integer	Date of the last file status change (calendar format)
43	FILE OWNER NAME OWNER	String	File owner
44	FILE OWNER ID UID	Integer	File owner identifier
45	FILE GROUP NAME FILEGROUP	String	File group

Table 4-1. FINFO Specifications (continued)

Number	Alias	Data Type	Item Description
46	FILE GROUP ID GID	Integer	File group identifier
47	FILE TYPE FILETYPE	String	File type
-47	FILE TYPE INTEGER INTFILETYPE	Integer	File type
48	RECORD TYPE RECTYPE	Integer	Record type
49	BYTE FILE SIZE BYTEFILESIZE	Integer	Current file size (in bytes)
50	KSAM VERSION KSAMVERS	Integer	KSAM XL file version
51	KSAM LABEL KSAMPARAM	some type by reference	KSAM XL parameters
52	DEVICE TYPE DEVTYPE	String	MPE/iX device type
-52	DEVICE TYPE INTEGER INTDEVTYPE	Integer	MPE/iX device type
53	RELEASED	Boolean	Secured/Released

The FINFO function is often used in file maintenance routines to create, delete, or determine certain characteristics of a file. The following examples demonstrate the responses to several FINFO requests about the file X.PUB.SYS.

```
:CALC FINFO('X',0)
TRUE
:CALC FINFO('X','EXISTS')
TRUE


:CALC FINFO('X',13)
ASCII, FIXED, NOCCTL, STD
:CALC FINFO('X','FMTFOPT')
ASCII, FIXED, NOCCTL, STD

:CALC FINFO('X',1)
X.PUB.SYS

:CALC FINFO('X','FULLFNAME')
X.PUB.SYS

:CALC FINFO('X',38)
/SYS/PUB/X
```

The last example asks for the absolute pathname of the file "X" (HFS syntax). Compare this with the example immediately preceding it, which asks for the file name in MPE syntax.

Comparing Results

The comparison operators, equal (=), less than (<), greater than (>), not equal to (<>), less than or equal to (<=), and equal to or greater than (>=), are provided in the CI expression evaluator. These expressions produce a boolean result that can be used to set the criteria for a conditional branch or a looping structure.

Expression Substitution

Variables are dereferenced implicitly when used in the SETVAR, IF, WHILE, and CALC commands. Variables within an expression are also implicitly dereferenced. A variable can be implicitly dereferenced in any command, therefore, by including it in an expression.

The result of an expression, however, must be dereferenced explicitly. To do this, the expression is enclosed in brackets and preceded by an exclamation point. By explicitly dereferencing the expression and using implicit dereferencing of variables within the expression, the expression is more readable. The following examples demonstrate the use of explicit and implicit dereferencing when using expressions.

```
:BUILD X;REC=![-2*40]
```

```
:BUILD Y;DISC=![FINFO(FILE1,'EOF')*110]

:ECHO ![UPS(FILE1)] does not exist.
```

Expressions are evaluated from left to right within the operator hierarchy. This sequence can be altered, however, by including parentheses in the expression. Parentheses can be nested to provide a specific sequence for evaluation.

The following example shows the expression that centers the variable *TEXT* on an 80-character line. Note that *TEXT* is implicitly dereferenced within the brackets and explicitly dereferenced outside the brackets.

```
:ECHO ![RPT(" ",(80-LEN(TEXT))/2)]!string
```

In this example, a string is centered on an 80-character line by repeating the spacing character a number of times and concatenating the original string to it. Note the use of parentheses to override operator hierarchy and compute results in the proper sequence.

1. Evaluates the length of the string named *TEXT*.
2. Subtracts this length from the width of the line (80 characters).
3. Divides the remaining number of characters by 2.
4. Repeats the spacing character the specified number of times.
5. Dereferences the expression results.
6. Concatenates the expression results to the original string.

5

Using Language Constructs Available with CI

The CI programming structures provide standard programming routines for input, output, looping, and branching. These structures provide the basis for sophisticated programming. Several of the most common structures are described and demonstrated in the following sections. (Chapter 7 provides complete command file routines that have been excerpted in this chapter.)

Obtaining Input

Input to a command file or UDC can be supplied through parameter specifications entered when the command file or UDC is issued. Interactive prompts and responses can be embedded in the command file to receive input from the caller. Predefined command input can also be retrieved from an input file.

Identifying Parameters

PARM statements can be inserted in a command file or UDC to define parameter data that is entered when the command file or UDC is issued. The following example shows the command input to invoke the `FILINFO` command file. The file name `A` is provided when the command file name is issued.

```
FILINFO A
```

If parameter data is desired, one or more PARM statements can be included in the command file. For UDCs, parameters can be defined using the UDC name line, PARM statements, or both. Each entry defines one or more required or optional parameters. The following example defines the parameter input, a file name, in a PARM statement. The subsequent FINFO statement uses the parameter value to verify the existence of the named file. Note that a parameter must be explicitly dereferenced in all commands and expressions.

```
PARM FNAME
IF NOT (FINFO("!FNAME","EXISTS")) THEN
⋮
```

In this example, the file name is a required parameter since no default value has been provided for the parameter field. In processing the command file, the file name is inserted in any expression where the parameter `FNAME` is dereferenced, such as in the FINFO statement.

Note

Parameters must be accessed by explicit dereferencing. Implicit dereferencing will result in either a message stating that the variable does not exist or insertion of a variable value of the same name.

Parameter data is optional if a default value is provided in the PARM statement or UDC name line. The following example of the `LIST` command file defines six parameters and supplies default values for five of them. Note that the two PARM statements are used to differentiate between required and optional parameters.

```
PARM F1
PARM F2=$NULL,F3=$NULL,F4=$NULL,F5=$NULL,F6=$NULL
```

This example allows up to six file names to be entered when the command file is invoked. The first file name is the only required parameter. The value `$NULL` is substituted as the default value for the optional parameters. The following example shows how the `LIST` command file could be invoked.

```
LIST A,B,C
```

Each of the three parameters is matched with its associated parameter name. The file name `A` is identified as `F1`, the first parameter. File name `B` is `F2`, and file name `C` is `F3`. Because no other parameters were supplied in the call, `F4`, `F5`, and `F6` contain the default value `$NULL`. This method of coding is often used in looping constructs to provide input to multiple iterations of a process with a single invocation.

In a UDC, parameter data can be specified in two ways. Parameters can be defined in the UDC header on the line containing the UDC name, as shown in the following example.

```
LIST F1,F2=$NULL,F3=$NULL,F4=$NULL,F5=$NULL,F6=$NULL
```

Parameters can also be entered on the line after the UDC name and identified by the PARM keyword, as in command files.

```
LIST
PARM F1,F2=$NULL,F3=$NULL,F4=$NULL,F5=$NULL,F6=$NULL
```

Note that both of these methods can be used within a single UDC, as shown in the following example.

```
LIST F1
PARM F2=$NULL,F3=$NULL
PARM F4=$NULL,F5=$NULL,F6=$NULL
```

Prompting for Input

The INPUT command prompts the user for input data and reads user input from $STDIN, the user's terminal, into a defined variable. The data received with the INPUT command is considered string, even if it consists of numeric data.

The INPUT command allows three parameters. The first parameter specifies the variable name for the input data. The second specifies a character string to be used as the prompt. The length of time allowed for the user to enter input data can be specified by entering a third parameter.

In the following example, the echoed statements and the prompt for the user's selection create a menu of run variations. The user's selection at the prompt triggers a branch to the appropriate routine.

```
ECHO Enter 0 to exit.
ECHO       1 to review database.
ECHO       2 to update database.
INPUT CHOICE,"Which number do you select?",15
```

In the preceding example, the options in the ECHO commands and the prompting phrase from the INPUT command are displayed at the user's terminal. The variable, CHOICE, receives the user's response, to be analyzed in later processing. Fifteen seconds (,15) specifies the length of time to wait for a reply.

Retrieving Input from a File

By using redirection indicators, most CI statements can retrieve input from a file rather than from parameter data or direct user input. The name of the file containing the input data is preceded by the redirection indicator for input (<) and added to the appropriate statement. Only a single file name can be specified for input redirection; wildcard characters, therefore, cause errors. The following example shows how the editor is invoked and its input file specified in a single statement.

```
:EDITOR <INSFILE
```

When input is redirected, the file must exist. Unless otherwise specified in a file equation, the CI searches for a TEMP file first, then a permanent file. By default, it opens the file for read and share access. These defaults can be overridden by using a backreference to a file equation.

Redirection supports HFS (hierarchical file system) names. `ECHO Hi There! >> ./somefile` appends to the file `./somefile`

The following example receives processing input by redirecting the INPUT command's source from the user's terminal to a file named INFILE. The first record of INFILE is read into the variable named *RECORD*.

```
INPUT RECORD <INFILE
```

Because of possible confusion with the less than (<) character, redirection cannot be performed with the IF, WHILE, CALC,

SETVAR, REMOTE, or COMMENT statements. All other CI statements can perform input redirection.

Branching After Evaluation

Sequence within a command file or UDC can be controlled with the IF, ELSEIF, ELSE, and ENDIF statements. When the conditions specified in the IF statement are met, the actions following this statement are performed. When the conditions are not met, an optional ELSEIF or ELSE statement provides an alternative. Each IF construct must be ended with its associated ENDIF statement.

IF statements can be nested, providing subsequent levels of conditions. When nested, however, each IF statement must be ended by a matching ENDIF statement. The following example is from the `MENU` command file shown in the previous example. In this segment, the user's response to the input prompt is evaluated to run the appropriate database processing program.

```
IF CHOICE = "1" THEN
  DBREVIEW
ELSE
  IF CHOICE = "2" THEN
    DBUPDATE
  ENDIF
ENDIF
```

The ELSEIF statement provides a combination of the ELSE and IF statements and simplifies a nested IF construction. The `MENU` command file could have been written using the ELSEIF construction. The following example demonstrates how the ELSEIF statement simplifies the branching construct in the `MENU` command file.

```
IF CHOICE = "1" THEN
  DBREVIEW
ELSEIF CHOICE = "2" THEN
  DBUPDATE
ENDIF
```

Creating Processing Loops

The CI provides a looping structure through the WHILE and ENDWHILE statements. Combined with the expression evaluator functions, a series of statements can be repeated as long as a particular condition exists. The MENU command file provides a processing loop that continues prompting for the user's response and selecting the proper subroutine until the user selects 0 to exit the routine.

```
SETVAR CHOICE " "
WHILE CHOICE <> "0" DO
  ECHO Enter 0 to exit.
  ECHO       1 to review database.
  ECHO       2 to update database.
  INPUT CHOICE,"Which number do you select?",15
  IF CHOICE = "1" THEN
    DBREVIEW
  ELSEIF CHOICE = "2" THEN
    DBUPDATE
  ENDIF
ENDWHILE
```

Often an index is set and evaluated for processing elements of an array. Although the CI does not support data arrays or tables, variable names can be constructed so that a portion of the name can be incremented to access several variables in a sequence, similar to array or table processing.

In the following example, a simulated array of data has been created by appending sequential numbers to a common variable name. The variables range from RM_ARRAY1 to RM_ARRAYn, depending on the number of elements needed. In this example, the variable RM_ARRAY0 contains the number of elements used in the sequence.

Each variable in this example is accessed in turn by concatenating the common portion of the variable name with an incremented index variable, creating the sequence of variable names. The processing loop continues to access consecutive variables until the index exceeds the value of RM_ARRAY0.

```
SETVAR RM_INDEX 1
WHILE RM_INDEX <= RM_ARRAY0 DO
  ECHO Purging !"RM_ARRAY!RM_INDEX" now.
  PURGE !"RM_ARRAY!RM_INDEX"
  SETVAR RM_INDEX RM_INDEX+1
ENDWHILE
```

Reporting Results

Usually, output from a command file or UDC is displayed on the user's terminal. The ECHO command and, in some situations, the PRINT command display output from a command file or UDC to the terminal. In some instances, output can be used as input for a subsequent procedure. This is easily accomplished by redirecting output from a command file or a UDC to a file.

Displaying Output to the Terminal

The ECHO command displays information to the user's terminal. All text following the command name is displayed. Variables and expressions can be dereferenced to insert their values or results in the display.

```
ECHO FOR FILE !FILE:
ECHO RECSIZE: ![FINFO(FILE,14)], EOF: &
     ![FINFO(FILE,19)], FLIMIT: ![FINFO(FILE,12)].
```

The dereferenced values are substituted into the string to be displayed on the user's terminal. The following example shows the result of this ECHO command for the file A.PUB.SYS.

```
FOR FILE A.PUB.SYS:
RECSIZE: -80, EOF: 5, FLIMIT: 5.
```

The PRINT command accepts the MPE escape syntax that permits you to access HFS (hierarchical file system) files and directories. Refer to *New Features of MPE/iX: Using The Hierarchical File System* (32650-90351).

Redirecting Output to a File

By using redirection indicators, most CI statements can send output to a file rather than display it on the user's terminal. A redirection specification for an output file is added to the appropriate statement. The name of the file to receive the output is preceded by the redirection indicator (either > or >>). Only a single file name can be specified for a single output redirection specification; wildcard characters, therefore, cause errors. The following example shows how the output of the SHOWCATALOG command is sent to the file UDCNAMES.

```
:SHOWCATALOG >UDCNAMES
```

In the previous example, the redirection indicator, >, creates a new, temporary file to which the SHOWCATALOG output is sent. The file is created with the following default characteristics:

- 80-byte, variable length, ASCII records.
- DISC=10000.
- TEMP file.
- NOCCTL.

The file is created with write, shared, and multiaccess. These default values can be overridden by using a backreference to a file equation. Note that if the file exists and the output redirection indicator is >, the output from the statement overwrites any existing data in the file.

To open an existing file and append new output, the output redirection indicator is >>. The following example creates a file for the SHOWME command output. It then appends output from the SHOWOUT command to the same file.

```
:PURGE ABC
:SHOWME >ABC
:SHOWOUT >>ABC
```

In this example, the file ABC is created, and the SHOWME output is written to it. The file is then closed. It is reopened for append access to receive output from the SHOWOUT command.

Output redirection works differently if the output destination is a device file, such as a spoolfile or tape file. The same commands as in the previous example are sent to a spoolfile in the following example:

```
:FILE lp602;DEV=EPOC;ENV=lp602.hpenv.sys
:SHOWME >*lp602
:SHOWOUT >>*lp602
```

As with the prior example, the spoolfile is created based on specifications in the file equation, and the output from the SHOWME command is written to it. The spoolfile is then closed and becomes ready to print. A second spoolfile is created for the SHOWOUT command output and is opened with append access. A spoolfile is completed and ready to print when it is closed; it cannot be reopened. The append access of the SHOWOUT command is unnecessary in this case.

Returning to Calling Environment

The RETURN command exits a command file or UDC and returns to the calling process, whether it is the CI, an application, or another UDC or command file. This provides a mechanism for creating an error routine. The following routine tests that a parameter is a positive integer. If not, an error message is displayed, and control is returned to the calling environment.

```
PARM COUNT, P2=" ", P3=" "
IF TYPEOF(!COUNT) <> 1 OR !COUNT < 0 THEN
  ECHO Expected a positive integer for COUNT
  RETURN
ENDIF
```

The ESCAPE command can also be used to exit a command file or UDC. Unlike RETURN, ESCAPE causes control to leave all UDCs or command files, regardless of nesting levels, and returns to the CI

prompt. If a jobstream invokes the command, the job is flushed from the system.

6

Accessing Variables and CI Commands in Applications

The intrinsic mechanism provides a means of communicating with the CI from an application program. Several intrinsics manipulate user-defined and predefined variables from within an application. User-defined and predefined variables can be set and retrieved within an application as a means of communicating with another process in the same session.

Intrinsics can also execute a CI command, UDC, or command file from within an application. This method can be used to supply system information to the application or to perform a routine more efficiently through CI commands.

Note that each programming language calls an intrinsic in a unique manner. Refer to the appropriate application language guide for details about calling intrinsics.

Using Intrinsics to Set Variables

Any user-defined variable can be set from within an application program using the `HPCIPUTVAR` intrinsic. Variables set in this way are available to any process within the specified session. The variable name specified in the intrinsic must be a valid variable name. An optional `STATUS` parameter can be specified to receive status information on the success of the intrinsic call.

Item number and item pairs identify the type of data that is to be stored in the variable. The item number and item pairs that are defined for the `HPCIPUTVAR` intrinsic are listed below:

0 Ignored by the intrinsic.

1 Indicates that the following parameter contains an integer value to be assigned to the specified variable name. (No other item number and item pair is needed.)

2 Indicates that the following parameter contains the string value to be assigned to the specified variable name. (Item number 11 is also required for string variables.)

3 Indicates that the following parameter contains a nonzero value if the variable value is to be boolean TRUE. It holds a zero if the value is to be boolean FALSE. (No other item number and item pair is needed.)

11 Indicates that the following parameter contains the length of the string value to be assigned to the variable name. (This

item number and item pair is required when item number 2 has been specified.)

14 Indicates that the following parameter contains a nonzero value if the string value specified by item number 2 is to be stored as entered. It contains a zero if the string value specified by item number 2 is to be interpreted. If interpreted, a string containing a number within the range of -2147483648 to 2147483648 is interpreted as a numeric value. A string containing TRUE or FALSE (upper or lower case) is interpreted as a boolean value.

It is possible to set variables on both a process local and a job or session global basis. For a list of variables whose modifications exist only for the locality of the process see Appendix A of the *MPE/iX Commands Reference Manual* 32650-90003.

If you wish to modify session or job level variables then the `HPCIGETVAR` and `HPCISETVAR` intrinsics should be called rather than calling the `COMMAND` intrinsic to execute the `SETVAR` and `SHOWVAR` commands. The `COMMAND` intrinsic `SETVAR` will not set a session level variable.

The following example uses the `HPCIPUTVAR` intrinsic to set a variable named `ANS`. It also loads it with numeric data (1) contained in the parameter `ANS_INPUT`. Status on the success of the `HPCIPUTVAR` intrinsic call is returned in the parameter named `STATUS`.

```
HPCIPUTVAR(ANS,STATUS,1,ANS_INPUT)
```

The next example specifies that the variable named `ANS2` is loaded with data from `ANS_INPUT`. The length of the string is specified in `ANS_LEN`. Note that the `STATUS` parameter has been omitted, signified by the two commas after the variable name.

```
HPCIPUTVAR(ANS2,,2,ANS2_INPUT,11,ANS2_LEN,14,ANS2_INT)
```

The data type of the variable in this example depends on the contents of the `ANS2_INT` parameter. If this field contains a nonzero character, the data is considered string character data. If the `ANS2_INT` field contains a zero, the input, `ANS_INPUT`, is interpreted to determine whether the variable contains numeric or boolean data.

The following sample program creates a variable named `MYVAR` using the `HPCIPUTVAR` intrinsic.

```
Procedure Create_CI_Variable;
Const
  KeyWord_StringValue  = 2;   { keyword #2 in the intrinsic manual }
  KeyWord_StringLength = 11;  { Keyword #11 in the intrinsic manual }
  StringLength         = 34;
  CIVarNameLen         = 6;   { Length of variable name to be created }
Type
  StatusType = Record
                 Case Boolean Of
                   True  : ( Error_Num : Integer );
                   False : ( Info,
                             SubSys : ShortInt );
               End;
  CIVarNameType   = Packed Array[ 1..CIVarNameLen ] Of Char;
  ToBeWrittenType = Packed Array[ 1..StringLength ] Of Char;
  CIVarValueType  = ToBeWrittenType;
Var
  Status       : StatusType;
  CIVarName    : CIVarNameType;
  KeyValue_ActualStringContents  : CIVarValueType;
  KeyValue_ActualStringLength    : Integer;
  Procedure HPCIPUTVAR;    INTRINSIC;
  Procedure TERMINATE;     INTRINSIC;
  Procedure Check_Status( Var Status : StatusType );
  Begin {Check_Status}
    With Status Do
      If Info <> 0 Then
         Begin
           Writeln( 'Subsystem Number: ', SubSys );
           Writeln( 'Info            : ', Info );
           TERMINATE;
         End;
  End;  {Check_Status}
Begin {Create_CI_Variable}
  CIVarName  := 'MyVar ';    { Name of the variable to be created }
  KeyValue_ActualStringContents := 'Programmatically Created Variable';
  KeyValue_ActualStringLength   := StringLength;
  { Create MyVar variable. }
  HPCIPUTVAR( CIVarName, Status,
              KeyWord_StringValue, KeyValue_ActualStringContents,
              KeyWord_StringLength, KeyValue_ActualStringLength );
  Check_Status( Status );
End;  {Create_CI_Variable}
Begin {Main}
  Create_CI_Variable;
End.  {Main}
```

Figure 6-1. HPCIPUTVAR Intrinsic Example

The HPCIPUTVAR intrinsic can also be used to modify the value of a predefined variable. The predefined variable to be modified is identified by specifying the variable name in the intrinsic call. Its value is determined by item number and item pairs specifying the data type and the value to be inserted. Note that some predefined variables cannot be modified. (Refer to the *MPE/iX Commands Reference Manual* (32650-90003) for a list of predefined variables that can be modified.)

The HPCIDELETEVAR intrinsic deletes any user-defined variables within the appropriate session. The variable to be deleted is specified by its variable name. Wildcard characters can be used to delete multiple variables. The STATUS parameter returns the status of the intrinsic call to the calling program. Note that no predefined variables can be deleted. The following example deletes the variable named ANS.

```
HPCIDELETEVAR(ANS,STATUS)
```

The PUTJCW intrinsic sets and loads a job control word (JCW), a 16-bit unsigned integer variable. The SETJCW intrinsic sets only the predefined variable named JCW. Both of these functions can be accomplished with the HPCIPUTVAR intrinsic by specifying the appropriate variable name and loading a numeric value within the range of 0 to 65,535. To ensure program readability, use the HPCIPUTVAR intrinsic in all cases.

Using Intrinsics to Retrieve Variables

The HPCIGETVAR intrinsic returns the current value of the specified variable from the session variable table. HPCIGETVAR requires only a variable name. The STATUS parameter is optional. If specified, the success of the intrinsic call is returned in the STATUS parameter.

Item number and item pairs identify the parameters to receive the retrieved value. The item number and item pairs that are defined for the HPCIGETVAR intrinsic are listed below:

0 Ignored by the intrinsic.

1 Indicates that the following parameter receives the integer value of the variable. If the variable is not an integer, a zero is returned.

2 Indicates that the following parameter receives the string value of the variable. If the variable is not a string, ASCII zero is returned.

3 Indicates that the following parameter receives a 1 if the value is the boolean TRUE or a 0 if the value is the boolean FALSE. If the variable is not a boolean, a 0 is returned.

10 Indicates that the following parameter contains the length of the byte array receiving the variable's string value. If a length is passed and a byte array is not, an error is returned.

11 Indicates that the following parameter receives the actual length (in bytes) of the variable's string value.

12 Indicates that the following parameter contains a nonzero value to dereference the variable recursively. A zero value in this parameter retrieves the level-1 value of the variable.

13 Indicates that the following parameter receives a 1 if the variable found is an integer, 2 if the variable found is a string, or 3 if the variable found is a boolean.

The following example retrieves the current value of the string variable `ANS`. The `STATUS` parameter returns the status of the intrinsic call. The parameter `ANS_LEN` returns the length of the string retrieved.

```
HPCIGETVAR(ANS,STATUS,2,ANS_STRING,11,ANS_LEN);
```

When you are unsure of the data type of the variable to be retrieved, several parameters can be included in the intrinsic call. The value is retrieved and loaded into the appropriate parameter field based on its type. The variable type parameter (13) is also specified to identify the actual data type and, therefore, the location of the retrieved value. The following example identifies parameters for each data type and a variable type parameter to identify the data type of the retrieved value. Note that the `STATUS` parameter has been omitted in this intrinsic call.

```
HPCIGETVAR(ANS2,,1,ANS2_INTEG,2,ANS2_BYTE,3,ANS2_BOOL,10,ANS2_LEN,13,ANS2_VARTYPE);
```

Another method of retrieving a variable value when the data type is not known is shown in the following example. Initially, the `HPCIGETVAR` intrinsic call is executed to determine only the data type of the variable. The retrieved data type is then used to branch to the appropriate intrinsic statement to retrieve the variable value. In the following example, the variable type is retrieved and analyzed. If the `ANS2_VARTYPE` parameter contains 1, the second intrinsic call is used to retrieve the integer value.

```
HPCIGETVAR(ANS2,STATUS,13,ANS2_VARTYPE);
⋮
HPCIGETVAR(ANS2,STATUS,1,ANS2_INTEG);
```

The following program sample retrieves the logical device number of the user's terminal.

```
Function At_Physical_Console : Boolean;
Const
  PhysicalConsoleLDev     = 20;
  KeyWord_GetIntegerValue = 1; { keyword #2 in the intrinsic manual }
  CIVarNameLen            = 9;
Type
  StatusType = Record
                 Case Boolean Of
                   True  : ( Error_Num : Integer );
                   False : ( Info,
                             SubSys : ShortInt );
               End;
  CIVarNameType  = Packed Array[ 1..CIVarNameLen ] Of Char;
  CIVarValueType = Integer;  { See constant KeyWord_GetIntegerValue }
Var
  Status              : StatusType;
  CIVarName           : CIVarNameType;
  KeyValue_CIVarValue : CIVarValueType;
  Procedure HPCIGETVAR;    INTRINSIC;
  Procedure TERMINATE;     INTRINSIC;
  Procedure Check_Status( Var Status : StatusType );
  Begin {Check_Status}
    With Status Do
      If Info <> 0 Then
         Begin
           Writeln( 'Subsystem Number: ', SubSys );
           Writeln( 'Info            : ', Info );
           TERMINATE;
         End;
  End;  {Check_Status}
Begin {At_Physical_Console}
  CIVarName  := 'HPLDEVIN ';  { Retrieve the ldev number associated }
                              { with this terminal }
  { Read the contents of HPLDEVIN variable. }
  KeyValue_CIVarValue := 0;
  HPCIGETVAR( CIVarName, Status,
              KeyWord_GetIntegerValue, KeyValue_CIVarValue );
  Check_Status( Status );
  At_Physical_Console := KeyValue_CIVarValue = PhysicalConsoleLDev;
End;  {At_Physical_Console}
Begin {Main}
  If At_Physical_Console Then
     Writeln( 'Execution of this application on the console is not allowed.' )
  Else
     Writeln( 'Not on console -- run the application.' );
End.  {Main}
```

Figure 6-2. HPCIGETVAR Intrinsic Example

The FINDJCW intrinsic retrieves the value of a specified JCW variable. The GETJCW intrinsic retrieves the current value of only the predefined variable named JCW. Both of these functions can be accomplished with the HPCIGETVAR intrinsic by specifying the appropriate variable name. To ensure program readability, use the HPCIGETVAR intrinsic in all cases.

Using Intrinsics to Execute CI Commands

The HPCICOMMAND intrinsic provides programmatic access to the CI command set, command files, and UDCs. Most CI commands can be invoked from an application program by calling this intrinsic and specifying the command name and appropriate parameters. (Commands that cannot be called from an application program include: ABORT, BYE, CHGROUP, DATA, DO, EOD, EOJ, EXIT, HELLO, JOB, LISTREDO, OPTION, REDO, RESUME, and SETCATALOG.)

The command and its parameters are passed as a command string through the first parameter of the HPCICOMMAND intrinsic. This command string must be terminated by a carriage return character. A prompt character is not included. Any error code set by the command is returned in the command error parameter.

The HPCICOMMAND can be used to invoke a command file or UDC from an application program. This allows multiple commands or frequently used routines to be called with a single intrinsic reference.

The COMMAND intrinsic functions similarly to the HPCICOMMAND intrinsic. Its command set, however, is limited to the MPE/iX built-in commands. You cannot access UDCs or command files with this intrinsic. In most instances, the HPCICOMMAND intrinsic is preferred because of its ability to execute multiple commands through UDCs and command files.

7

Sample Command Files

Samples of the command files that have been excerpted in the previous text are provided in the following section. They provide examples of how the CI's programming elements can be used to create sophisticated sequential routines. An explanation of each routine's purpose, a sample of its use, and highlights of the routine are supplied.

To Center a String

The CENTER command file is an example of string manipulation functions. It centers a supplied string and displays it on $STDLIST, usually the user's terminal screen.

A character string is entered as parameter data when the command file is invoked. The variable CENT_SPC is defined as a string variable of 40 blank spaces. A portion of this string of blanks is inserted on the left side of the original text string to center it. The expression determines the number of blanks to be entered and adds them to the character string before echoing the phrase. Note that parameters must always use explicit dereferencing.

```
PARM STRING
SETVAR CENT_SPC "                                        "
ECHO ![LFT(CENT_SPC,(80-LEN("!STRING"))/2)]!STRING
DELETEVAR CENT_SPC
COMMENT  ** END OF CENTER **
```

Figure 7-1. Center Command File

The repeat function can also be used to simplify this routine.

```
PARM STRING
ECHO ![RPT(" ",(80-LEN("!STRING"))/2)]"!STRING"
COMMENT  ** END OF CENTER **
```

Figure 7-2. Center Command File with the Repeat Function

To Set a Function Key

The FKEY command file sets a single function key. It demonstrates setting escape sequences with string concatenation and variable dereferencing.

Parameter input defines the key to be set by the command file (KEY), the label (L1), the string to be generated when the function key is pressed (S1), and the key attribute parameter (A1). The possible entries for the key attribute parameter are:

0 = NORMAL

1 = LOCAL

2 = TRANSMIT

The commands contained in FKEY are provided in the following example. (Note that an ampersand is used to continue a lengthy command on a second line. The CI will concatenate the first and second lines to execute the command.)

```
PARM KEY,L1=" ",S1=" ",A1=2
IF HPJOBTYPE="S" AND HPDUPLICATIVE THEN
  ECHO ![CHR(27)]&f!"A1"a!"KEY"k&
       ![LEN"!L1"]d![LEN("!S1")]L!L1!S1
  ECHO ![CHR(27)]&jB
ENDIF
COMMENT ** END OF FKEY **
```

Figure 7-3. Function Key Command File

To set the F1 key to perform a test and display a message, the following command string would be entered.

```
FKEY 1,test,'echo this is a test'
```

Once the command string has been entered, pressing the F1 key results in the following display:

```
:echo this is a test
this is a test
```

Figure 7-4. FKEY Sample Output

To Add User Capabilities

The ADDCAP command file adds capabilities to a user's capability list. The ALTUSER command is used to alter the capability list. The AM capability, therefore, is required to execute this command file. Since the new capability does not become effective until the user logs on again, the user is offered the option of being logged on automatically. This command also permits an authorized user to change the UID of a user. Refer to *New Features of MPE/iX: Using The Hierarchical File System* (32650-90351).

```
PARM CAP=""
IF ("!CAP"="") THEN
  ECHO (ADDCAP): Your capabilities are: !HPUSERCAPF.
  RETURN
ENDIF
IF (POS(UPS("!CAP"),HPUSERCAPF) <> 0) THEN
  ECHO (ADDCAP): You already have    : !CAP.
  ECHO (ADDCAP): The capabilities are: !HPUSERCAPF.
  RETURN
ENDIF
SETVAR CIERROR 0
CONTINUE
ALTUSER !HPUSER;CAP=![HPUSERCAPF + ",!CAP"]
IF CIERROR <> 0 THEN
  ECHO (ADDCAP): The capabilities remain: !HPUSERCAPF.
ELSE
  ECHO (ADDCAP): !HPUSER new capabilities are: ![HPUSERCAPF + ",!CAP"].
  SETVAR ADDCAP_TEMP "N"
  INPUT ADDCAP_TEMP,"(ADDCAP): Log off/on now (Y/N) ==>",10
  IF HPCIERR = -9003 THEN
    COMMENT ** TIMED READ EXPIRED **
    ECHO
    ECHO (ADDCAP): Timed 10-second read expired.  &
                   Logon cancelled.
  ELSEIF NOT(UPS(LFT(ADDCAP_TEMP,1)) = "Y") THEN
      ECHO (ADDCAP): New capabilities take effect at next &
                     logon.
  ELSE
    HELLO !HPJOBNAME,!HPUSER.!HPACCOUNT,!HPGROUP
  ENDIF
ENDIF
DELETEVAR ADDCAP_TEMP
COMMENT ** END OF ADDCAP **
```

Figure 7-5. Additional Capability Command File

An error routine is triggered by the contents of CIERROR. If an unacceptable parameter is entered, the CIERROR field is updated and the capabilities remain as they were.

There are three possible outcomes in running this command file using acceptable input:

- No capability was entered as a parameter value. The user's current capabilities are listed.
- The user already has the capability.
- The capability is added to the user's list. Note that a relogging option prompts the user to relog automatically or to wait until the next logon to activate the new capabilities.

The following examples illustrate these three possibilities.

```
:ADDCAP
(ADDCAP): Your capabilities are: AM,BA,IA


:ADDCAP AM
(ADDCAP): You already have    : AM
(ADDCAP): The capabilities are: AM,BA,IA


:ADDCAP DS
(ADDCAP): SAMPLE new capabilities are: AM,BA,IA,DS
(ADDCAP): Log off/on now Y/N ==>

(ADDCAP): Timed 10-second read expired.  Logon cancelled.
```

Figure 7-6. ADDCAP Sample Output

To Retrieve File Information

The `FILINFO` command file displays file label information for a given file. This command file provides an extensive routine that analyzes the file name if the file is not found. A fully qualified file name is returned even if the parameter input was not qualified.

```
PARM FILE
IF NOT (FINFO("!FILE","EXISTS")) THEN
  COMMENT ** FILE DOES NOT EXIST **
  IF LFT("!FILE",1) <> "*" AND LFT("!FILE",1) <> "$" THEN
    COMMENT **QUALIFY FILE BEFORE REPORTING NON-EXISTENCE**
    IF POS(".","!FILE") > 0 THEN
      COMMENT ** A GROUP NAME IS SPECIFIED **
      IF POS(".","!FILE",2) > 0 THEN
        COMMENT ** FILE NAME IS FULLY QUALIFIED **
        ECHO ![UPS("!FILE")] does not exist.
      ELSE
        ECHO ![UPS("!FILE")].!HPACCOUNT does not exist.
      ENDIF
    ELSE
      ECHO ![UPS("!FILE")].!HPGROUP.!HPACCOUNT &
           does not exist.
    ENDIF
  ELSE
    ECHO !FILE does not exist.
  ENDIF
  RETURN
ENDIF
COMMENT ** FORMAL FILE DESIGNATOR **
ECHO (FINFO): Full file description for &
              ![FINFO("!FILE",1)] follows:
COMMENT ** CREATOR AND CREATE/MODIFY DATES **
ECHO    Created by ![FINFO("!FILE",4)] on &
            ![FINFO("!FILE",6)].
ECHO    Modified on ![FINFO("!FILE",8)] at &
            ![FINFO("!FILE",24)].
COMMENT ** FILE CODE **
IF FINFO("!FILE",9) = "" THEN
  ECHO    FCODE: ![FINFO("!FILE",-9)].
ELSE
  ECHO    FCODE: ![FINFO("!FILE",9)] &
                 (![FINFO("!FILE",-9)]).
ENDIF
COMMENT ** RECORD SIZE, END OF FILE, FILE LIMIT **
ECHO   RECSIZE: ![FINFO("!FILE",14)], EOF: &
       ![FINFO("!FILE",19)], FLIMIT: ![FINFO("!FILE",12)].
COMMENT ** FILE OPTIONS **
SETVAR _FOPT FINFO("!FILE,-13)
ECHO FOPTIONS: ![FINFO("!FILE",13)] (#!_FOPT, &
       ![OCTAL(FOPT)], !HEX(_FOPT)]).
DELETEVAR _FOPT
```

Figure 7-7. File Information Command File

This command file searches for a designated file. If found, its formal file designator, creator and creation date, modification information, file code, record size, end of file, file limit, and file options are displayed. The following example provides a sample of the display provided by the FILINFO command file.

```
:FILINFO SAMPLE
(FINFO): Full description for SAMPLE.PUB.MILL follows:
   Created by CLM on WED, MAY 10, 1989.
   Modified on WED, MAY 10, 1989 at 3:21 PM.
   FCODE: 0.
   RECSIZE: -80, EOF: 5, FLIMIT:5.
   FOPTIONS: ASCII, FIXED, NOCCTL, STD (#5, %5, $5).
```

Figure 7-8. FILINFO Sample Output

If the file is not found, an extensive routine determines the fully qualified file name under which the search was performed. This routine determines if the specified file name is a backreferenced file name or a system-defined file name. It also determines whether a group or account designator was included in the original specification. If necessary the group or account are added to the file name to fully qualify it in a response to the user.

To Create a Calculator

The CALCIT command file provides the user with an interactive calculator. The user is prompted for an equation to be solved or an MPE/iX command to execute. To exit CALCIT, the user enters a carriage return.

Note that the CENTER command file is called to center the display headings on the user's terminal. After an equation is solved, the result is displayed on the terminal using escape sequences to append the result to the original input line and to underline it. The predefined variable HPMSGFENCE is used to suppress error messages.

The CALCIT prompt is based on the command file name. By dereferencing the latest command entered in the history stack (!-1), the name that called this command file becomes a part of the prompt. If the command file name is changed to COMPUTE, for example, the prompt automatically becomes COMPUTE ==>.

```
PARM ENH_CH=D
COMMENT Interactive calculator using calc and input
ECHO ![CHR(27) + "h" + CHR(27) + "J"]
CENTER "MPE/iX INTERACTIVE CALCULATOR"
CENTER ": executes any MPE/iX command!"
CENTER "Type [RETURN] to exit"
ECHO
SETVAR CALCIT_ESC CHR(27) + "A" + RPT(CHR(27) + "C", 64)
SETVAR CALCIT_PROMPT LFT(UPS("!-1"), POS(' ',"!-1" + ' ')-1) + ' ==> '
WHILE SETVAR (CALCIT_EXPR, RTRIM(INPUT(CALCIT_PROMPT))) <> '' DO
  COMMENT Save length before trimming leading blanks
  SETVAR CALCIT_LEN LEN(CALCIT_EXPR)
  SETVAR CALCIT_EXPR LTRIM(CALCIT_EXPR)
  IF LFT(CALCIT_EXPR,1) = ":" THEN
    CONTINUE
    ![RHT(CALCIT_EXPR,LEN)CALCIT_EXPR)-1)]
  ELSE
    SETVAR HPMSGFENCE 2
    SETVAR CIERROR 0
    CONTINUE
    SETVAR HPRESULT !CALCIT_EXPR
    SETVAR HPMSGFENCE 0
    IF CIERROR <> 0 THEN
      COMMENT User entered an invalid expression.
      COMMENT Display error.
      ECHO ![RPT(' ',LEN(CALCIT_PROMPT) + HPCIERRCOL-17)]^
      ECHO !HPCIERRMSG
    ELSE
      ECHO ![LFT(CALCIT_ESC, 2 + (LEN(CALCIT_PROMPT) + CALCIT_LEN) * 2) +&
           ' = ' + CHR(27) + '&d!ENH_CH']!HPRESULT
    ENDIF
  ENDIF
ENDWHILE
DELETEVAR CALCIT_@
```

Figure 7-9. Calculator Command File

The following example shows the terminal display to calculate the equation 5 + 7.

```
:CALCIT
                        MPE/iX INTERACTIVE CALCULATOR
                         : executes any MPE/iX command
                            Type [RETURN] to exit

CALCIT ==> 5+7 = 12
CALCIT ==>
:
```

To Create a Menu of Options

The MENU command file provides a mechanism for running two programs, DBREVIEW and DBUPDATE. The user's response of 1 or 2 performs the proper program and prompts the user for another selection. A response of 0 to exit the routine ends the WHILE loop and ends the command file execution.

```
SETVAR CHOICE ""
WHILE CHOICE <> "0" DO
  ECHO Enter 0 to exit
  ECHO       1 to review database
  ECHO       2 to update database
  INPUT CHOICE,"Which number do you select?",15
  IF CHOICE = "1" THEN
    DBREVIEW
  ELSEIF CHOICE = "2" THEN
      DBUPDATE
  ENDIF
ENDWHILE
```

Figure 7-10. Menu Command File

To List Multiple Files

The LIST command file prints the contents of multiple files to device class LP. Up to six files can be specified as parameters when invoking the command file.

```
PARM F1,F2=$NULL,F3=$NULL,F4=$NULL,F5=$NULL,F6=$NULL
SETVAR LIST_I 1
SETVAR LIST_F "!!F1"
WHILE LIST_I <= 6
  IF UPS("!LIST_F") <> "$NULL"
    FILE !LIST_F;DEV=LP
    ECHO (LIST): Printing of !LIST_F is in progress.
    PRINT !LIST_F,*LIST_F
    RESET !LIST_F
    SETVAR LIST_I LIST_I+1
    SETVAR LIST_F "!!F!LIST_I"
  ENDIF
ENDWHILE
DELETEVAR LIST_I, LIST_F
```

Figure 7-11. List Command File

8

Command Input/Output Redirection (CIOR)

Command Input/Output Redirection (CIOR) enables you to define different files for command input and command output. Without CIOR, command input and output defaults to $STDIN or $STDLIST. For sessions, $STDIN and $STDLIST are your terminal. For jobs, $STDIN and $STDLIST are spoolfiles although $STDLIST is most commonly seen as a printed job output (spoolfile) listing.

CIOR provides independent management of redirection by the command interpreter. This means that command executors, user commands (user defined commands (UDCs) and command files) need not be modified to take advantage of CIOR. Redirection is available from sessions, jobs, programmatically (via the COMMAND and HPCICOMMAND intrinsics) and in break mode. CIOR provides defaults for redirection file attributes; however, you can override the defaults with file equation backreferences.

Redirection files are written to the temporary file domain unless this default is overidden with a file equation. (Other defaults are covered later in this section).

Redirecting Command Input and Output

Redirection is accomplished with the use of the following simple redirection indicators <, > or >>, followed by a file name, or backreference to a file equation.

The < sign indicates that CI input comes from the specified file. The > sign indicates that CI output goes to the specified file. The >> sign also indicates that CI output goes to the specified file; however, output is appended to the file (if it exists) so existing data is not overwritten.

Redirection is supported for single files only; therefore, wildcarding on the file name is meaningless and returns an error.

The following are examples of CIOR.

Redirecting Command Input

To redirect command input you would enter:

```
command < infile
```

Command can be any MPE/iX command except for the following:

- CALC
- COMMENT
- ELSEIF
- IF
- REMOTE
- SETVAR
- SETJCW
- TELL
- TELLOP
- WARN
- WHILE

and `infile` is the file that contains the input to that command.

For example:

```
editor < edinput
```

This invokes the EDITOR and instructs it to read data from the file edinput. The data is used by EDITOR as if it was being typed as input at the terminal.

Redirecting Command Output

To redirect command output you would enter:

```
command > outfile
```

Once again, command is as described above and `outfile` is the file which contains the output of the command.

For example:

```
showme > userfile
```

This invokes the `showme` command and instructs it to write its output to a file called `userfile`.

Redirecting Both Command Input and Output

CIOR enables you to redirect both input and output at the same time.

To accomplish this, you would enter:

```
command <infile >outfile
```

For example,

```
editor <edinput >$null
```

This invokes EDITOR, instructs it to read input from file `edinput` and to write output to `$NULL`. Output from EDITOR for includes the banner, the prompt and all other EDITOR output.

Redirecting I/O with a File Backreference

Backreferencing a file equation with CIOR is both simple and useful. To accomplish it you enter:

```
file formaldesig1;parm ... ;parm ...
file formaldesig2;parm ... ;parm ...
command <*formaldesig1 >*formaldesig2
```

Note

The `FILE` command accepts the MPE escape syntax, allowing you to reference HFS (hierarchical file system) file names, but only on the right hand side of the equation. Refer to *MPE/iX Commands Reference Manual* (32650-90003) for more information about using the `FILE` command with HFS names.

You may backreference both input and output to different file equations or either input or output to a single file equation.

For example:

```
file sourclst;save;rec=-80,,f,ascii
listfile S@, qualify > *sourclst
```

This example first establishes file equations to override redirection file defaults. It then issues a `listfile` command and directs its output to the file sourclst. The `;save` option in the file equation keeps the file in the permanent domain. Without `;save` the file sourclst would stay in the default (temporary) file domain.

Redirection file defaults are discussed in a later section in this chapter.

Note

The `LISTFILE` command accepts the MPE escape syntax, allowing you to reference HFS (hierarchical file system) file names, but only on the right hand side of the equation. Refer to *MPE/iX Commands Reference Manual* (32650-90003) for more information about using the `LISTFILE` command with HFS names.

Appending Redirected Command Output

When the > sign is used, command output is redirected to the specified file and it begins writing output at the beginning of the file. When the >> sign is used, command output is redirected to the specified file, but it is appended to the file, so information already contained in the file is not overwritten. If >> is used and the file does not exist, it will be created.

The append form of output redirection (>> sign) is useful when the output is a logging message or has a logging function, or when the user is creating a file with several commands. If your goal is to create a file containing only the output of the current command you should use the > sign and not the >> sign.

Redirection supports HFS (hierarchical file system) names. `ECHO Hi There! >> ./somefile` appends to the file `./somefile`

To append redirected output enter:

```
command >>outfile
```

For example:

```
echo text file1 > edinput
echo find "patt1" >> edinput
echo delete >> edinput
echo keep >> edinput
echo exit >> edinput
```

The first line in this example produces the temporary file `edinput`. The next four lines append edit commands to this file. The result is a file of commands that might be read by the EDITOR subsystem with CIOR in the following command:

```
editor <edinput >$null
```

This command invokes the EDITOR and causes it to read the commands previously written to the file `edinput`. Command output, in this case, is redirected to `$NULL`.

Printing the `edinput` file shows the commands that the EDITOR would read as input:

```
print edinput
text file1
find "patt1"
delete
keep
exit
```

Redirecting Output to a Device File

To redirect output to a device file (such as a printer), simply backreference a file equation for the device as follows:

```
file formaldesig;dev= ...
command >*formaldesig
```

For example:

```
file lp602;dev=epoc;env=lp602.hpenv.sys
listfile @.@.myaccount,-2 > *lp602
```

This example would send a listing of all access control definition (ACD) data to the printer defined by the file equation with formal file designator `lp602`.

The Append Option with Device Files

The redirection append option (use of the >> sign) produces a different result with device files than with disk files. For example, you could enter:

```
file lp602;dev=epoc;env=lp602.hpenv.sys
listfile @.@.myaccount,-2 > *lp602
listfile @.@.myaccount,4 >> *lp602
```

Both `listfile` commands are directed to a printer in this example. When the first `listfile` command finishes, its spoolfile is closed and made ready for printing. Once this happens to a spoolfile it cannot be reopened. For this reason, the second `listfile` command cannot append to the first file even though the >> sign has been used. The second `listfile` command simply writes another spoolfile.

Stacked I/O Redirection

I/O redirection may be done even if I/O has already been redirected.

For example, say you have created a command file named `SHOW`, which contains the following:

```
parm showdest="$STDLIST"
showme
showout >!showdest
showvar
showjob >>!showdest ;job=!hpuser.!hpaccount
```

Now let's say you set a file equation for printer `LP602` and invoke `SHOW` as follows:

```
show outfile >*lp602
```

After variable substitution and I/O redirection, this results in the following sequence of commands and output destinations:

showme	output to lp602 (spoolfile)
showout	output to a temporary file named outfile
showvar	output to lp602 (a new spoolfile)
showjob	output appended to the temporary file outfile

Output from the `showme` and `showvar` commands goes to the device defined by the file equation for `lp602`.

Output from the `showout` command is redirected to the file `outfile` and output from `showjob` is appended to `outfile`. This occurs because the entire `show` command file has its output redirected to LP602. But inside the command file, the `showout` and `showjob` commands redirect output (again) to the temporary file name `outfile`.

Things to Remember about Redirection Constructions

When creating redirection constructions it is important to remember that the redirection specification is stripped from the command line *after* string substitution (variable and expression substitution) but *before* the command is actually executed.

For example, suppose you entered:

```
editor <myfile >$null newfile
```

With CIOR, this command works without problems because both redirection specifications, `<myfile` and `$null`, are removed from the command line before it is invoked. This leaves:

```
editor newfile
```

Input, however, would be read from `myfile` and command output would be sent to `$null`. An offline listing would go to `newfile` however, because that's the function of this parameter in the EDITOR subsystem.

To further illustrate how string substitution works, let's revisit an example already used in this chapter which has a command file with the following contents:

```
parm showdest="$stdlist"
showme
showout >!showdest
showvar
showjob >>!showdest ;job=!hpuser.!hpaccount
```

The default value for the `showdest` parameter is `$stdlist` in this example. If the user does not enter a parameter the `showout` command becomes

```
showout >$stdlist
```

The redirection specification is stripped from the command line and no redirection is done because none is necessary. `$STDIN` behaves similarly. (In fact the capability to specify `>$STDLIST` and `<$STDIN` as redirection specifications is provided to handle this type of defaulting).

But this example also illustrates why the scan for redirection specifications is done *after* string substitution. If the redirection scan were done before string substitution, or at the same time, the user couldn't specify an input or output redirection file using parameters or variables as in the last example. This ordering always allows the following:

```
setvar dest ">abc"
listf !dest
```

Because string substitution occurs first `listf` becomes `listf >abc` and then redirection sends the output to file `abc`.

Expression substitution is also done in the string substitution pass and would, therefore, also be performed before the redirection scan. For example, the `LISTF` command might be invoked as follows:

```
listsf ![input('ENTER THE FILESET TO BE DISPLAYED:')],6 &
>![input('ENTER THE FILE NAME FOR THE OUTPUT OF THE LISTF:')]
```

During the string substitution scan the user is prompted first for the file set, and then for the output file name. If the user entered `abc@ef` and then `outfile` the command line would be the following after the string substitution pass:

```
listf abc@ef,6 > outfile
```

Next, the output would be redirected and then the `listf` would be invoked.

Escaping Redirection

If, for any reason, you want to use the IO redirection indicators <,> or >> without having them function as such, you can precede them with the `!` sign.

For example, suppose you wanted to construct a command file to explain how to use I/O redirection which contained the following `echo` command:

```
echo To redirect $STDLIST use the construct, >filename.
```

This would cause `filename` to be a newly-created, temporary file containing the string preceding it.

To prevent this from happening, insert `!` before `>filename`, as follows:

```
echo To redirect $STDLIST use the construct !>filename
```

The resulting display would be:

```
To redirect $STDLIST use the construct >filename
```

The `!` can be used in the same way to escape the other redirection indicators (< and >>) also.

Redirection File Defaults

If a file equation is not used to specify the characteristics of an output redirection, file the following are taken as defaults:

256-byte, variable length, ASCII records

DISC=10000

Temporary file domain

NOCCTL

When output is redirected using the > filename redirection specification to a temporary file which *already* exists, the following defaults apply to the open:

- WRITE
- SHARED access
- MULTI access

If the >>filename specification is used instead, APPEND access is requested instead of WRITE.

A default also exists for input redirection. A file to which input has been redirected is opened by first attempting to open a temporary file with the name specified. This makes default input redirection consistent with default output redirection. If no such temporary file exists, an attempt is made to open a permanent file with that name.

When input is redirected, the file must exist and is opened with the following defaults:

- Look for a temporary file first, then an old permanent file
- READ, SHARED access

Determining Redirection: HPSTDIN and HPSTDLIST

CIOR provides two predefined MPE/iX string variables to reflect the state of I/O redirection. These variables can be used within User Commands or programs that need to detect if their input or output has been redirected. HPSTDIN will default to the string "$STDIN". HPSTDLIST will default to the string value "$STDLIST". These values also indicate whether or not input or output is currently redirected. If input is redirected, HPSTDIN will contain the name of the file to which input has been redirected. If input has been redirected through a file equation backreference, HPSTDIN will contain the formal file designator of the file equation. Similarly, HPSTDLIST will contain the filename or formal file designator of the file or file equation to which output has been redirected.

A

Predefined Variables

Various abbreviations are used in the type column of the following table to distinguish a variable's type and characteristics. The data types are identified as follows:

I - Integer format.
B - Boolean format (TRUE/FALSE).
S - String (ASCII) format.

No predefined variables can be deleted. The following abbreviations specify whether a variable can be modified or not:

R - Read only variable (cannot be modified).
W - Read/write variable (can be modified).

Any variables that cannot be altered or retrieved programmatically through the `HPCIGETVAR` or `HPCIPUTVAR` intrinsics are identified by the notation (NP). (Note that these variables can be accessed through the `HPCICOMMAND` and `COMMAND` intrinsics.)

Job control words are identified in the following table by the abbreviation JCW. They may be considered integer variables with legal values ranging from 0 to 65,535 and with bits 16 and 17 (bit 0 being the leftmost bit of 32 bits) having special interpretations. (For example, if bit 16 is set, the JCW setting is `FATAL`.)

Table A-1. Predefined Variables

Variable	Type	Definition	Initial Value
CIERROR	W JCW	last CI error number	zero
HPACCOUNT	R S	user's account name	logon account
HPACCTCAP	R I	current account capability mask	logon account capabilities
HPACCTCAPF	R S	current account capability formatted, for example, "AM, AL, GL, ND, SF, BA, IA"	formatted logon account
HPAUTOCONT	W B (NP)	enables (TRUE) / disables (FALSE) the automatic CONTINUE feature	FALSE
HPCIDEPTH	R I	number of nested CIs	1(=Root CI)
HPCIERR	W I	contains the most recent CI related error; similar to CIERROR except warnings are negative numbers and errors are posistive numbers	zero
HPCIERRCOL	W I	contains the column number of the offending parameter for the most recent CI command error	zero
HPCIERRMSG	R S	textual message for the most recent CIERROR (length of message is 0 for nonexistent CIERROR values)	(null)
HPCMDNUM	R I (NP)	current command sequence number	1
HPCMDTRACE (subject to change in future releases)	W B (NP)	enables (TRUE) /disables (FALSE) the User Command Tracing facility	FALSE
HPCMEVENTLOG	W I	when set to *n* $STDLIST displays the following *n* occurrences of tos/reg trap	zero
HPCONNMINS	R I	current session connect time in minutes	zero
HPCONNSECS	R I	current session connect time in seconds	zero
HPCONSOLE	R I	LDEV of the console	console ldev at logon
HPCONTINUE	R B (NP)	CI's continue state: FALSE=inactive, TRUE=active	FALSE
HPCPUNAME	R S	name of computer model, for example, "SERIES 960"	name of your logon computer model
HPCPUMSECS	R I	from root CI = current *session* CPU time in milliseconds; from other CI or process = current *process* CPU time in milliseconds	zero
HPCPUSECS	R I	from root CI = current *session* CPU time in seconds; from other CI or process = current *process* CPU time in seconds	zero

Table A-1. Predefined Variables (continued)

Variable	Type	Definition	Initial Value
HPCWD	R S	The HPCWD variable displays your current working directory name in the HFS (Hierarchical File System) convention. Because your current working directory is not shown by the SHOWME command, you may want to display it in your prompt by executing this command: `SETVAR hpprompt "!!hpcwd:"`	HFS syntax of logon group
HPDATE	R I	current day of month	logon day of the month
HPDATEF	R S	current formatted date	logon date
HPDAY	R I	current day of the week (1=SUNDAY)	logon day of the week
HPDTCPORTID	R S	port id of data terminal	null string
HPDUPLICATIVE	R B	duplicative (TRUE)/ nonduplicative (FALSE)	as appropriate
HPERRDUMP	W I (NP)	number of errors to be dumped from process error stack	zero
HPERRSTOLIST	W B	destination to which errors are to be written: TRUE=$STDLIST; FALSE=$STDERR	TRUE
HPEXECJOBS	R I	number of jobs and sessions currently in EXEC (executing) state	number of jobs and sessions in EXEC state
HPFILE	R S	contains the fully qualified file name of the currently executing command file or UDC file. If you are not in a UDC or command file, HPFILE returns an empty string.	empty string
HPFSERR	W I	file system error number for last CI command to generate file system error or warning	zero
HPGROUP	R S	current group name	logon group name
HPGROUPCAP	R I	current group capability mask	logon group caps
HPGROUPCAPF	R S	current group formatted capability mask, for example, "IA,BA,PH"	logon group caps
HPHGROUP	R S	home group name	home group
HPHOUR	R I	current hour number (24-hour clock)	logon hour
HPINBREAK	R B (NP)	FALSE=not in BREAK, TRUE=in BREAK mode (includes process BREAK and rit BREAK)	FALSE
HPINPRI	R I	input priority	logon input priority

Table A-1. Predefined Variables (continued)

Variable	Type	Definition	Initial Value
HPINTERACTIVE	R B	interactive (TRUE)/ noninteractive (FALSE)	as appropriate
HPINTRODATE	R S	formatted job/session logon date	date of logon
HPINTROTIME	R S	formatted job/session logon time	time of logon
HPJOBCOUNT	R I	number of jobs executing	logon number of executing jobs
HPJOBFENCE	R I	fence value for waiting jobs	logon jobfence
HPJOBLIMIT	R I	current job limit	job limit at logon
HPJOBNAME	R S	name of current job/session	logon job name
HPJOBNUM	R I	job/session number, for example, 12	your job/session number
HPJOBTYPE	R S	"S"=session, "J"=job	your job type
HPLDEVIN	R I	LDEV number for $STDIN	logon input LDEV
HPLDEVLIST	R I	LDEV number for $STDLIST	logon output LDEV
HPMINUTE	R I	current minute number	logon minute
HPMONTH	R I	current month number	logon month
HPMSGFENCE	W I (NP)	fence for the level of error messages printed by the CI: 0=errors/warnings, 1=errors only, 2=no error/warning messages	0=all errors and warnings are printed
HPNCOPIES	R I	number of $STDLIST copies for jobs	copies subparm of the outclass= parm of the JOB command
HPOUTCLASS	R S	output device class	logon output device class
HPOUTFENCE	R I	output fence value	logon output fence value
HPPATH	W S	search path for command files and implied RUN	"!hpgroup,pub, pub.sys"
HPPIN	R I	returns the process identification number of the currently executing process.	empty string
HPPROMPT	W S	CI's prompt string	":" (colon)
HPQUIET	R B	boolean indicating if session is accepting messages: FALSE = accepting messages; TRUE = not accepting messages	TRUE
HPREDOSIZE	W I	number of entries in the CI's redo stack	20
HPRESULT	W S, W I, or W B	value of the most recent CALC command evaluated (for example, "abc", 12, TRUE)	zero

Table A-1. Predefined Variables (continued)

Variable	Type	Definition	Initial Value
HPSCHEDJOBS	R I	number of jobs currently in SCHED state (scheduled state)	number of jobs in SCHED state
HPSESCOUNT	R I	number of sessions executing	logon number of sessions executing
HPSESLIMIT	R I	current session limit	session limit at logon
HPSTDIN	R S	file name or formal file designator to which input has been redirected, names for $STDIN	$STDIN
HPSTDLIST	R S	file name or formal file designator to which output has been redirected, names for $STDLIST	$STDLIST
HPSUSAN	R S	unique serial number assigned at the factory to each system for use by software	unique serial number assigned to your system at manufacture
HPSUSPJOBS	R I	current number of jobs in SUSP state (suspended)	numbers of jobs in SUSP state at logon
HPSYSNAME	W S	name of computer system (user-definable)	null string (" ")
HPSYSTIMEOUT	R I	reads from the sysgen configuration file what the minimum timeout is on the system. Works with HPTIMEOUT which controls the CI timed reads. HPSYSTIMEOUT is a minimum cap on HPTIMEOUT. HPTIMEOUT cannot be less than HPSYSTIMEOUT.	zero, means unlimited time.
HPTIMEF	R S	current formatted time	logon time
HPTIMEOUT	W I	number of minutes for CI reads (HPTIMEOUT=0 means no timeout)	zero
HPTYPEAHEAD	W B	boolean indicating if typeahead is turned on	FALSE
HPUSER	R S	current user name	logon user
HPUSERCAP	R I	current user's capability mask	logon user caps
HPUSERCAPF	R S	current user's formatted capability mask, for example, "IA,BA,PH"	logon user caps
HPUSERCMDEPTH	R I (NP)	number of nested UDCs and/or command files	zero
HPUSERCOUNT	R I	current number of licensed users accessing the system	licensed users
HPUSERLIMIT	R I	legal (licensed) limit of concurrent users allowed access to the system, can be accessed programmatically	legal user limit

Table A-1. Predefined Variables (continued)

Variable	Type	Definition	Initial Value
HPVERSION	R S	MPE XL version ID (*v.uu.ff*)	current MPE XL version
HPWAITJOBS	R I	current number of jobs waiting	number of jobs waiting at logon time
HPYEAR	R I	last two digits of the current year	logon year number
JCW	W JCW	job control word (variable)	zero

B

Evaluator Functions

The following table identifies most of the expression evaluator functions. For a list of the latest evaluator functions, refer to the *MPE/iX Commands Reference Manual* (32650-90003).

Table B-1. Expression Evaluator Functions

Symbol	Function	Example	Result
+(numeric)	addition	4 + 5	9
+(string)	concatenate	"abc" + "de"	abcde
-(numeric)	subtraction	12 - 6	6
-(string)	deletion of first occurrence	"abc" - "b"	ac
*	multiplication	4 * 5	20
/	integer division	79/ 10	7
^	exponentiation (0^0 yields 1)	2^3	8
either " or '	string identifier	either "abc" or 'abc'	abc
()	parentheses	(3 + 4) * 2	14
<	less than (Strings are compared character by character, until an inequality exists.)	5 < 6 'abcc' < 'abdc' 'abcd' &> 'abc'	TRUE TRUE TRUE
<=	less than or equal	"abc" <= "abc"	TRUE
>	greater than	"xyz" > "abc"	TRUE
>=	greater than or equal	"abc" >= "abc"	TRUE
<>	not equal	5 <> 6	TRUE
=	equal	“xyz”= “xyz”	TRUE
ABS(*integer*)	absolute value	ABS(-4)	4
ALPHA(*string*)	check if a string is alphabetic	ALPHA('abcd') ALPHA('ab3d ef')	TRUE FALSE
ALPHANUM(*string*)	check if a string is only alphabetics and digits	ALPHANUM('abCd') ALPHANUM('45abd') ALPHANUM('3d ef')	TRUE TRUE FALSE
AND	logical and	7=7 AND 5=5	TRUE

Table B-1. Expression Evaluator Functions (continued)

Symbol	Function	Example	Result
BAND	bitwise and	7 BAND 13	5
BNOT	bitwise not	BNOT 5	-6
BOR	bitwise or	5 BOR 2	7
BOUND(*varname*)	variable definition test (Returns TRUE if *varname* has been defined.)	BOUND(`HPPATH`)	TRUE
BXOR	bitwise exclusive or	7 BXOR 5	2
CHR(*integer*)	ASCII value (integer) ===> character	CHR(65)	A
CSL	circular shift left	-2 CSL 2	-5
CSR	circular shift right	-7 CSR 1	-4
DWNS(*string*)	shift string to lowercase (Operates on ASCII characters in ranges of 'a' through 'z' and 'A' through 'Z' only.)	DWNS('aBC&#dE')	abc&#de
FINFO(*filename,option*)	file information	FINFO('x.pub',0)	TRUE
HEX(*integer*)	convert to hexadecimal string	HEX(329)	$149
INPUT([*prompt*][,*wait*])	accept user input	INPUT('Enter choice:',20)	Enter choice: `Y` Return "Y"
LEN(*string*)	string length	LEN("abc")	3
LFT(*string, # chars*)	left string extraction	LFT('abc',2)	ab
LSL	logical shift left	7 LSL 1	14
LSR	logical shift right	-7 LSR 1	2,147,483,644
LTRIM(*string*[,*trimstr*])	trim left end of string	'X'+LTRIM(' abc') "X"+LTRIM(' ... abc', '.')	Xabc Xabc
MAX(*num1*[,*num2* ...])	find largest of several integers	MAX(5,4-3,70,0)	70
MIN(*num1*[,*num2* ...])	find smallest of several integers	MIN(5,4,-3,70,0)	-3
MOD	modulo	25 MOD 2	1
NOT	logical not	NOT(2>1)	FALSE
NUMERIC(*string*)	check is a string is all digits	NUMERIC('12345') NUMERIC('$a234ef')	TRUE FALSE
OCTAL(*integer*)	convert to octal string	OCTAL(329)	%511
ODD(*integer*)	determine if integer is odd	ODD(233) ODD(-2)	TRUE FALSE
OR	logical or	5=5 OR 2=3	TRUE

Table B-1. Expression Evaluator Functions (continued)

Symbol	Function	Example	Result
ORD(*string*)	ordinal	ORD('AbcD')	65
POS(*find str,source str*[,*n*])	find Nth occurrence of find str in source str (-N searches from right)	POS('ab','cgabd') POS('.','file.grp.acct',2) POS('.','file.grp.acct',-1)	3 9 9
RHT(*string, # chars*)	right string extraction	RHT("abc",2)	bc
RPT(*string,count*)	repeat a string (-count reverses string)	RPT('aBc',3) RPT('aBc',-3)	aBcaBcaBc cBacBacBa
RTRIM(*string*[,*trimstr*])	trim right end of string	RTRIM('abc ')+'X' RTRIM('abc ... ','.')+"X"	abcX abc X
SETVAR(*varname,expr*)	return result of *expr* and set *varname* to result	SETVAR(*myvar*,2*3+5)	sets variable *myvar* to 11 and returns 11
STR(*string,start pos, # chars*)	general string extraction	STR('abcde',2,3)	bcd
TYPEOF(*expression*)	type of variable or expression (0 = invalid, 1 = integer, 2 = string, 3 = Boolean value)	TYPEOF(`HPPATH`)	2 (string)
UPS(*string*)	shift string to uppercase (Operates on ASCII characters in range of 'a' through 'z' and 'A' through 'Z'.)	UPS('aBc5d')	ABC5D
XOR	logical exclusive or	7=7 XOR 5=5	FALSE

Index